Visualisierung von Audio-Daten in VR

Manuel-Philippe Hergenröder (12085)

Vertiefende Studienarbeit, WS 2019

Prof. Dr. Damon T. Lee

Ehrenwörtliche Erklärung

Hiermit versichere ich, dass die vorliegende Arbeit von mir selbstständig und ohne unerlaubte Hilfe angefertigt worden ist, insbesondere dass ich alle Stellen, die wörtlich oder annähernd wörtlich aus Veröffentlichungen entnommen sind, durch Zitate als solche gekennzeichnet habe.
Ich versichere auch, dass die von mir eingereichte schriftliche Version mit der digitalen Version übereinstimmt. Weiterhin erkläre ich, dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde/Prüfungsstelle vorgelegen hat. Ich erkläre mich damit nicht einverstanden, dass die Arbeit der Öffentlichkeit zugänglich gemacht wird. Ich erkläre mich damit einverstanden, dass die Digitalversion dieser Arbeit zwecks Plagiatsprüfung auf die Server externer Anbieter hochgeladen werden darf. Die Plagiatsprüfung stellt keine Zurverfügungstellung für die Öffentlichkeit dar.

Karlsruhe, der 16.03.2020	Manuel-Philippe Hergenröder	
Ort, Datum	Vorname Nachname	Unterschrift

1 Einleitung

Schon lange ist die Visualisierung von Informationen eine wichtige Methode der Vermittlung von Informationen. Dabei besitzt der Sehapparat die größte Bandbreite zum Gehirn im Vergleich zu den anderen Sinnesorganen. Im Gegensatz zum Beispiel zur akustischen Verarbeitung können eine große Menge an Informationen bzw. Daten „auf einem Blick“ verarbeitet werden. Insbesondere auch in Bezug auf den zeitlichen Verlauf, wohingegen beim Hören immer nur der gegenwärtige Moment bewusst erfasst werden kann. Das heißt die Anreicherung von gehörtem Ton mit visuellen Informationen kann unser Verständnis über die Struktur des Klangmaterials sinnvoll ergänzen.

2 Konzeption

Die angestrebte Softwarelösung soll ausgehend von Offline-Audio-Daten, die über eine Importfunktion aus einer Audiodatei heraus in einen Puffer geladen werden können, eine in VR „begehbare“ Visualisierung ermöglichen. Dafür soll eine frequenzbasierte Darstellung mithilfe der Fourier-Transformation verwendet werden, um Eigenschaften der Audio-Daten in visueller Form zu offenbaren, die evtl. beim bloßen Hören verborgen bleiben. Die Daten des Audio-Samples sollen dabei „im Ganzen“ dargestellt werden – also nicht nur eine Visualisierung des momentan abgespielten Ausschnitts. Durch die Nutzung von Kopfhörern und eines Head-Mounted-Display (HMD) soll eine Immersion hergestellt werden, welche aufgrund der Kombination von Hörsinn und Sehsinn das Erfassen und Erleben von Klanginformationen intensiviert. Die physische Bewegung in Form einer VR-Room-Scale-Experience dient einer intuitiven Navigation durch den virtuellen Raum.

3 Entwicklung der Visualisierung von Audio-Daten in VR

3.1 Hardwarevoraussetzungen

Für die Visualisierung in VR ist zum einen das HMD HTC VIVE inkl. zweier VIVE Controller erforderlich und zum anderen ein leistungsfähiger PC mit dedizierter 3D-

Grafikkarte. Die von HTC VIVE angegebenen Mindestanforderungen[1] setzen als CPU „Intel Core i5-4590/AMD FX 8350 oder besser“, als GPU eine „NVIDIA GeForce GTX 1060, AMD Radeon RX 480 oder besser“, „4 GB RAM oder mehr“ Arbeitsspeicher, „HDMI 1.4, DisplayPort 1.2 oder neuer“, „1x USB 2,0 oder höher“ und als Betriebssystem „Windows 7 SP1, Windows 8.1 oder später, Windows 10“ voraus. Dies ist insbesondere wichtig, da zu schwache Hardware für Bildaussetzer sorgt, was im Zusammenhang mit dem Head-Tracking zu Übelkeit und Kopfschmerzen führen kann. Im Idealfall sollten immer mind. 90 Bilder pro Sekunde ohne sogenannte Frame-Drops – also das Überspringen von Einzelbildern – gewährleistet sein.

Ohne HMD lässt sich die Visualisierung auch auf einem 2D-Monitor mit Tastaturnavigation und verringerter Immersion verwenden. Die Hardwareanforderungen fallen dabei etwas geringer aus, da in der Regel nur 60 Hz Bildwiederholrate erreicht werden müssen.

Für die schnelle Berechnung der Fourier-Transformation ist generell eine schnelle CPU mit guter Multi-Threading-Performance ratsam, um Wartezeiten zu verkürzen.

Bei dem System, welches zur Entwicklung verwendet wurde, handelt es sich um einen PC mit Intel i9-9900K, 32GB RAM und einer NVIDIA Geforce RTX 2070. Dies hat sich als adäquat herausgestellt.

3.2 Softwarearchitektur

Als Grundlage für die Visualisierung wurde die Laufzeit- und Entwicklungsumgebung Unity gewählt. Diese Spiele-Engine ist für nicht-kommerzielle Entwicklungen kostenlos nutzbar und bietet ein weitreichendes Ökosystem aus Assets, Erweiterungen und Anbindungen an externe Bibliotheken. Außerdem lässt sich das Verhalten durch selbstgeschriebene Skripte (u.a. in der .NET-Sprache C#) beliebig anpassen. Über das SteamVR-Plugin[2] aus dem Unity Asset Store lässt sich das Head-Mounted-Display HTC VIVE, für das dieses Projekt konzipiert ist, ansteuern.

[1] VIVE - “Wie sind die Systemanforderungen?” https://www.vive.com/de/support/vive/category_howto/what-are-the-system-requirements.html (letzter Abruf: 04.02.2020)
[2] Unity Asset Store – SteamVR Plugin - https://assetstore.unity.com/packages/tools/integration/steamvr-plugin-32647 (letzter Abruf: 04.02.2020)

Für den Import von Audio-Daten wird die Open-Source-Bibliothek NAudio[3] für .NET (entwickelt von Mark Heath) verwendet. NAudio hat sich durch den Open-Source-Charakter (lizensiert unter der Microsoft Public License – kurz Ms-PL) und die ausgereifte Implementation (Entwicklungsbeginn war 2002, der Code wird regelmäßig gepflegt, zuletzt 2020)[4] angeboten. Außerdem ist die Bibliothek komplett in C# .NET geschrieben – was die Einbindungen in Unity sehr einfach macht – und bietet eine vielfältige Unterstützung verschiedener Audio-Formate und Kompressions-Codecs. Zusätzlich könnte in Zukunft auch MIDI leicht in das Projekt eingebunden werden, da NAudio auch hierfür Klassen anbietet ohne eine zusätzliche Bibliothek einbinden zu müssen.

Das Zerlegen der Audio-Daten in seine Frequenzanteile wird mit der C-Bibliothek FFTW[5] realisiert. Ausschlaggebend dafür war die im Vergleich zu anderen Bibliotheken schnelle Performance[6], das freie Lizenzmodell, welches mit einer umfangreichen Dokumentation hergeht, sowie die Tatsache, dass FFTW viele verschiedene Algorithmen implementiert (u.a. verschiedene Varianten der diskreten und schnellen Fourier-Transformation – auch in umgekehrter Richtung). Dafür wurde nach sorgfältiger Abwägung auch die kompliziertere Einbindung einer C-Bibliothek in Unity in Kauf genommen. Für die Anbindung von FFTW in C# .NET wird der Wrapper FFTWSharp genutzt.[7]

Als verteilte Versionsverwaltung wird Git[8] auf einer selbst-betriebenen Gitlab-Installation, sowie Github, verwendet.

3.3 Bedienung

Das Programm VrUnitySpectrum.exe starten. Jetzt lässt sich eine Audio-Datei am PC-Monitor auswählen (siehe Abb. 1). Unter „Examples“ finden sich bereits einige Beispiele zum Testen – ansonsten können beliebige Mono-Dateien aus dem Dateisystem geladen werden.

---

[3] NAudio – Audio and MIDI library for .NET - https://github.com/naudio/NAudio (letzter Abruf: 04.02.2020)
[4] GitHub - naudio/NAudio: Audio and MIDI library for .NET – FAQ – https://github.com/naudio/NAudio#faq (letzter Abruf: 16.03.2020)
[5] FFTW Home Page – http://fftw.org/ (letzter Abruf: 04.02.2020)
[6] Comparison of FFT Implementations for .NET – https://www.codeproject.com/Articles/1095473/Comparison-of-FFT-Implementations-for-NET (letzter Abruf: 28.01.2020)
[7] C# wrapper for FFTW – https://github.com/tszalay/FFTWSharp (letzter Abruf: 05.02.2020)
[8] Git –fast-version control – https://git-scm.com/ (letzter Abruf: 04.02.2020)

Berechnung und Darstellung des Spektrums erfolgen nun (siehe Abb. 2 und 3).

Nach dem Aufsetzen des HMD kann nun mit dem linken VIVE-Controller zur Navigation teleportiert werden. Dafür die Mitte des Touchpads gedrückt halten und den Zielpunkt anvisieren (siehe Abb. 4).

Mit dem rechten VIVE-Controller lässt sich nun mit Drücken auf die Mitte des Touchpads die Wiedergabe starten und stoppen. Die Wiedergabe läuft in einer Schleife. Mit Druck auf den linken Bereich des Controllers lässt sich die Wiedergabe an den Anfang der Audiodatei setzen (Tastenbelegung siehe Abb. 5).

Alternativ lässt sich das Programm auch ohne HMD auf einem 2D-Monitor nutzen: Dabei kann mit den Tasten W, A, S, D eine Bewegung im virtuellen Raum durchgeführt werden. Mithilfe der Leertaste lässt sich die Wiedergabe starten und stoppen, mit Backspace die Wiedergabeposition zurücksetzen. Bei gedrückter rechter Maustaste kann die Blickrichtung mit der Maus verändert werden.

Die Spektrum-Daten werden während der Wiedergabe entsprechend der Abspielposition blau eingefärbt, um eine bessere Orientierung zu gewährleisten (siehe Abb. 6).

Mit der Tastenkombination „ALT+F4“ lässt sich das Programm beenden.

Innerhalb der Unity-IDE ist die Bedienung grundsätzlich identisch. Abweichend von der Stand-Alone-Variante erscheint jedoch kein Dialog zur Dateiauswahl. Die Audio-Datei wird über den Inspector des GameObjects „Audio“ (siehe Drop-Down-Menü „Select Audio Test File“) ausgewählt und das Starten und Beenden des Programms erfolgt über den Play-Button in der Unity-Oberfläche. Dies erleichtert das Testen des Programms während der Entwicklung.

3.4 Implementation und Fallstricke

Zunächst wurde SuperCollider[9] als Audio-Engine verwendet, was sich insbesondere für den Datenaustausch mit Unity als problematisch herausgestellt hat. Über die OpenSoundControl[10]-Spezifikation ließ sich der Datenaustausch in einer frühen Entwicklungsphase implementieren. Da das Protokoll allerdings eher für das Bereitstellen

[9] SuperCollider – https://supercollider.github.io/ (letzter Abruf: 04.02.2020)
[10] OpenSoundControl – http://opensoundcontrol.org/ (letzter Abruf: 04.02.2020)

von Steuerdaten und nicht für den Austausch großer Datenmengen konzipiert ist, musste ein eigener Handler zum Aufteilen der Audio-Daten in mehrere OSC-Messages implementiert werden. Problematisch erwies sich dabei vor allem der Verlust von einzelnen OSC-Messages, da OSC UDP-Pakete verwendet. Unter anderem musste eine Pause zwischen den OSC-Messages eingehalten werden, weil sonst – vermutlich durch Begrenzungen von Puffergrößen der OSC-Bibliothek und/oder des TCP/IP-Stacks – ein Großteil der Pakete nicht beim Empfänger (Unity) ankam. Außerdem wurden die Pakete durchnummeriert – ähnlich zu „sequence numbers" beim Transmission Control Protocol –, um einerseits die korrekte Reihenfolge der Daten zu gewährleisten und andererseits verlorengegangene Pakete identifizieren zu können. Insgesamt betrachtet ist die Methode OSC zu nutzen als langsam, fehleranfällig und aufwendig bzgl. der Implementation zu betrachten.

Ein anderes Problem – bezogen auf die Verwendung von SuperCollider – ist, dass es sich als komplex herausgestellt hat, auf die FFT-Rohdaten innerhalb von sclang zugreifen zu können, um sie dann via OSC zu Unity weiterzuleiten.[11] Die eigentlichen FFT-Daten liegen im Speicherbereich von scsynth. Ein Austausch über das Dateisystem oder einen Shared-Memory-Bereich wäre prinzipiell denkbar gewesen. Aufgrund der spärlichen Dokumentation und des hohen Aufwands wurde schlussendlich davon abgesehen SuperCollider als Audio-Engine zu verwenden.

Nach einem kompletten Rewrite setzt das Projekt auf die NAudio-Bibliothek für den Import der Audiodaten und als Playback-Engine. Dabei werden Mono-Daten aller gängigen Sampleraten, sowie 8, 16, 24, 32 oder 64bit Bittiefe unterstützt. Die Audio-Daten werden zunächst aus einer WAV PCM Audio-Datei eingelesen (durch die Nutzung von NAudio werden auch andere Containerformate und Codecs wie AIFF, MP3, etc. unterstützt).

Danach wird die Fouriertransformation sequenziell auf einzelne Blöcke der Audio-Daten angewendet. Die Ergebnisse bilden die Grundlage zur Generierung eines Meshes. Es wurde mit verschiedenen Darstellungsweisen experimentiert, u.a. einer zylindrischen, die die Werte der einzelnen FFT-Bins radial in Zackenform anordnet, was sich in der Praxis

[11] Anmerkung: SuperCollider besteht aus verschiedenen modularen Bestandteilen, u.a. dem Server „scsynth", welcher die eigentliche Audioverarbeitung durchführt, sowie beispielsweise dem Client „sclang" (oder andere Alternativen), welcher die Audio-Language implementiert hat, ausführt und via OSC mit dem Server kommuniziert. Siehe auch: http://doc.sccode.org/Guides/ClientVsServer.html (letzter Abruf: 16.03.2020)

als unergonomisch erwiesen hat, da dies zu vielen Kopfdrehungen und Bücken geführt hat. Letztendlich hat sich eine „Terraindarstellung“, also das Darstellen der Daten wie bei einer begehbaren Gebirgsformation, bewährt. Zusätzlich ist eine Teleportation möglich, um einerseits einen großen virtuellen Raum, der die Größe des physischen Raumes übersteigen kann, zu ermöglichen – und andererseits lange Laufwege zu vermeiden. Die Teleportation – in der Fachliteratur auch „Point & Teleport“ ist dabei auch einer der effektivsten Methoden, um Motion-Sickness zu vermeiden.[12] Dabei wird zunächst mithilfe eines Lichtstrahls ein Punkt anvisiert und eine Teleportation zu der neuen Position im virtuellen Raum findet unmittelbar statt – ohne sichtbare Translationsbewegung. Sogenannte Visually-Induced-Motion-Sickness[13] kann dann auftreten, wenn visuelle Bewegung nicht mit einer physischen Bewegung einhergeht, d.h. eine Diskrepanz im Gehirn entsteht zwischen den Bewegungen, die durch den optischen Sehreiz, und jenen, die beispielsweise durch Informationen des Gleichgewichtssinns oder des Bewegungsapparates vermittelt werden.

Das Mesh wird dabei iterativ in Unity zur Laufzeit generiert. Problematisch war zunächst das Rendern aller Daten innerhalb eines einzelnen Meshes. Unity nutzt standardmäßig nur 16bit für die Mesh-Indizes, so dass nur 65,536 vertices indexiert werden können. Dies lässt sich zwar mit dem Attribut .indexFormat auf 32bit umkonfigurieren[14], um alle Daten in einem einzelnen Mesh unterzubringen – allerdings führte dies zu einer extrem schlechten Performanz bis hin zum Absturz von Unity. Erst durch das Erzeugen individueller Meshes pro FFT-Block konnten hohe Frameraten und genügende Stabilität erreicht werden. Vermutlich hängt dies mit einer stark verbesserten Parallelisierbarkeit der Render-Pipeline zusammen.

Das Playback der Audio-Daten erfolgt über die NAudio-Bibliothek. Die in Unity integrierte Sound-Engine wird nur für die Wiedergabe der akustischen Teleportationsbestätigung genutzt.

---

[12] Vgl. Schäfer, Dmitrij – Untersuchung der Bewegungsmethoden im virtuellen Raum mit Bezug auf Cybersickness, S. 72

[13] Vgl. Jerald, Jason, NextGen Interactions. – *The VR Book – Human-Centered Design for Virtual Reality*, S. 163

[14] Anmerkung: Dazu wird nach dem Erstellen des Meshes die Property .indexFormat des Mesh-Objektes = UnityEngine.Rendering.IndexFormat.UInt32 gesetzt. Dies hat zur Folge, dass für den Index Buffer 32bit genutzt werden, so dass bis zu 4 Milliarden Vertices indexiert werden können. Standardmäßig werden 16bit genutzt, um Bandbreite und Speicher zu sparen und da nicht alle Plattformen mehr als 16bit unterstützen. Dadurch sind nur 65.635 Vertices pro Mesh möglich. Siehe auch https://docs.unity3d.com/ScriptReference/Mesh-indexFormat.html (letzter Abruf: 06.02.2020)

Die Bildraten sind auf dem Testsystem (siehe Kapitel 3.1 Hardwarevoraussetzungen) sowohl im 2D-Betrieb, als auch im stereoskopischen VR-Betrieb mit HMD stabil und hoch genug. Eine Analyse mit dem in Unity enthaltenen Profiler ergibt, dass das Projekt GPU-Bound ist – d.h. die CPU ist größtenteils nicht ausgelastet und wartet in der Regel auf die Bereitstellung des nächsten Bildes von der Grafikkarte (siehe Abb. 7). Die Berechnung der Fourier-Transformation, welche CPU intensiv ist, wird nur einmalig nach Auswahl der Audio-Datei ausgeführt. Danach liegt die meiste Last auf der Grafikkarte, welche (exemplarisch für das bereitgestellte Beispiel-Audio „sinesweep_1Hz_48000Hz_-3dBFS_30s.wav“) etwa 14 Millionen Vertices für das FFT-Spektrum zeichnen muss. Eine weitere Erklärung für die hohe Auslastung der Grafikkarte und weiteres Optimierungspotenzial liegt im benutzerdefinierten Shader für das Material des FFT-Spektrums: Für die Einfärbung der einzelnen Vertices von grün nach rot, um den FFT-Wert zu verdeutlichen, wurde ein Shader implementiert, welcher Farbwerte einzelner Vertices des Meshes beim Rendering berücksichtigt. Der Transfer dieser Farb-Daten vom Vertex-Shader zum Fragment-Shader innerhalb der Grafikkarte kostet – insbesondere bei der hohen Anzahl von Vertices – viel Zeit.[15] Hier besteht Potenzial zur Optimierung durch eine alternative Implementation, die keine Vertex-Colors verwendet.

4 Fazit, Ausblick und Möglichkeiten

Durch den Aspekt der Begehbarkeit der Audio-Daten entsteht eine Immersion und Zugänglichkeit, die das Erfassen des Klangmaterials vereinfacht. Die Navigation im virtuellen Raum erfolgt intuitiv, so dass der Fokus ganz auf der Darstellung der Audiodaten liegt – ohne, dass komplexe Bedienphilosophien verinnerlicht werden müssen. Einsatzmöglichkeiten bestehen insbesondere im Bildungssektor, da das Verständnis von frequenzbasierten Darstellungen spielerisch vermittelt wird.

Möglich wären auch das parallele Laden und Betrachten von mehreren Audio-Dateien, um Vergleiche ziehen zu können.

[15] Unity Forums: Performance implications between vertex colors vs textures, bgolus – https://forum.unity.com/threads/performance-implications-between-vertex-colors-vs-textures.671698/#post-4496353 (letzter Abruf: 04.03.2020)

Weitere Ideen wären u.a. weitere UI-Elemente, die mehr Informationen vermitteln, wie beispielsweise eine Anzeige des aktuellen Frequenzbereiches des anvisierten FFT-Bins oder Anzeige diverser Metadaten.

Insgesamt drängt sich bei der Benutzung der realisierten Lösung auch der Wunsch nach Bearbeitung der Audio-Daten auf. So könnten mit verschiedenen Werkzeugen im virtuellen Raum die Audio-Daten modelliert oder zerlegt und wieder zusammengesetzt werden – mit anschließendem Export durch rückwärtsgerichtete Fourier-Transformation. Dies könnte auch im künstlerischen Kontext zur Neuerschaffung von Klangmaterial durch Re-Modellierung einen Nutzen haben. So könnte eine Art Sandbox für die Bearbeitung von Klang geschaffen werden, welche durch die in VR herrschenden Bedien-Paradigmen einen neuen analytischen, künstlerischen sowie spielerischen Zugang zu den Audio-Daten ermöglicht.

5 Abbildungen

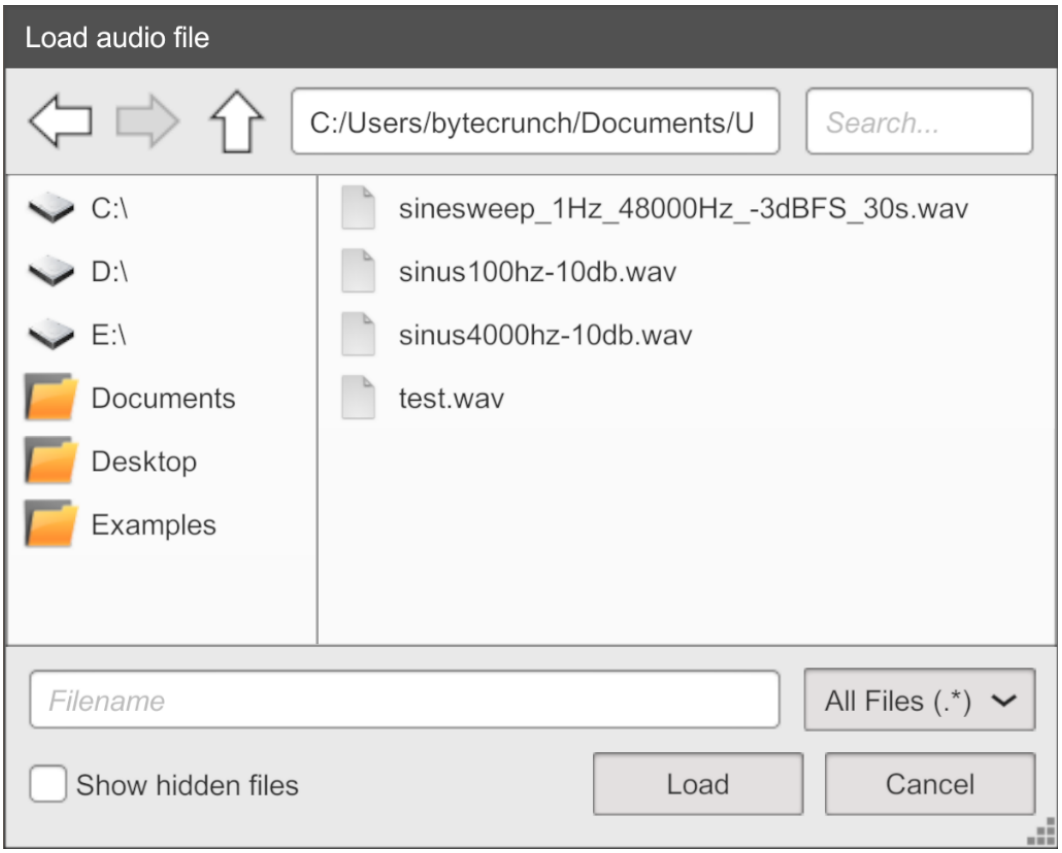

Abb. 1 – Dateiauswahl

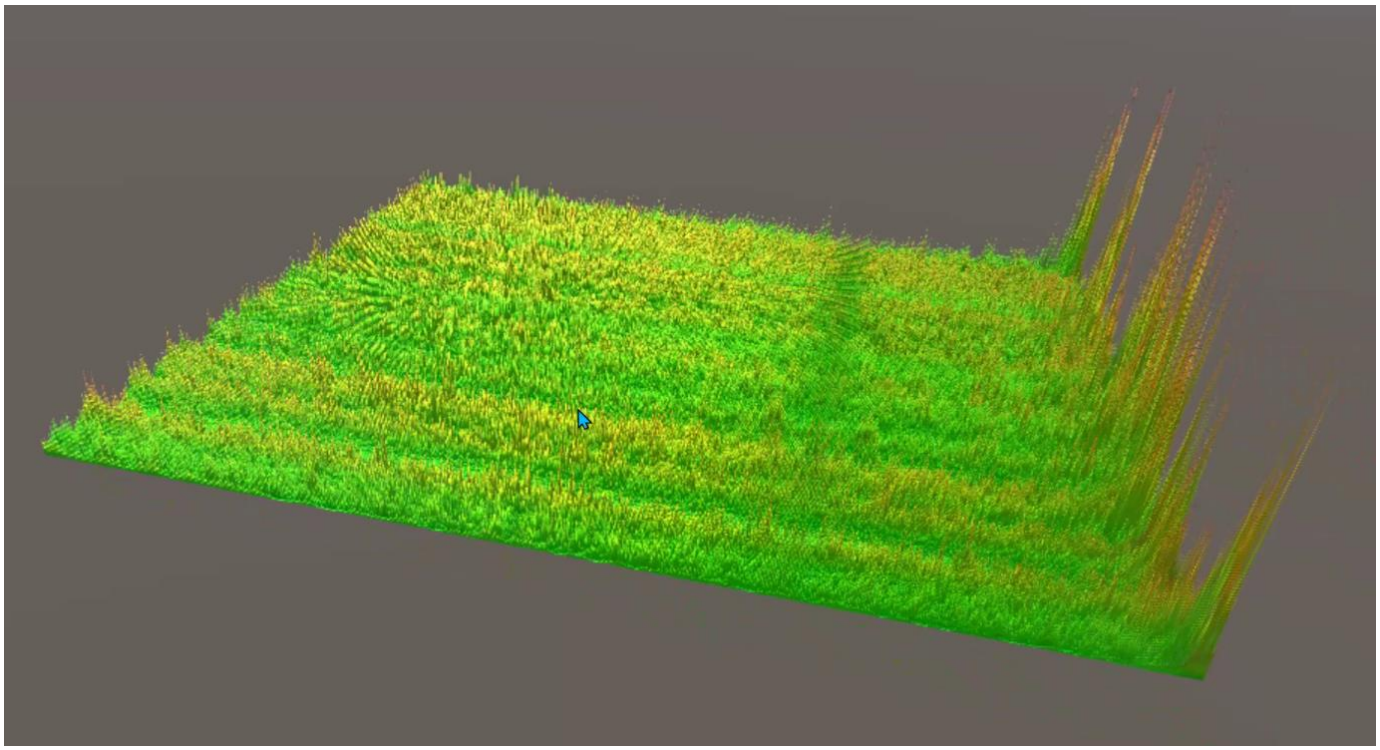

Abb. 2 – Anzeige des Spektrums nach Laden der Audio-Datei

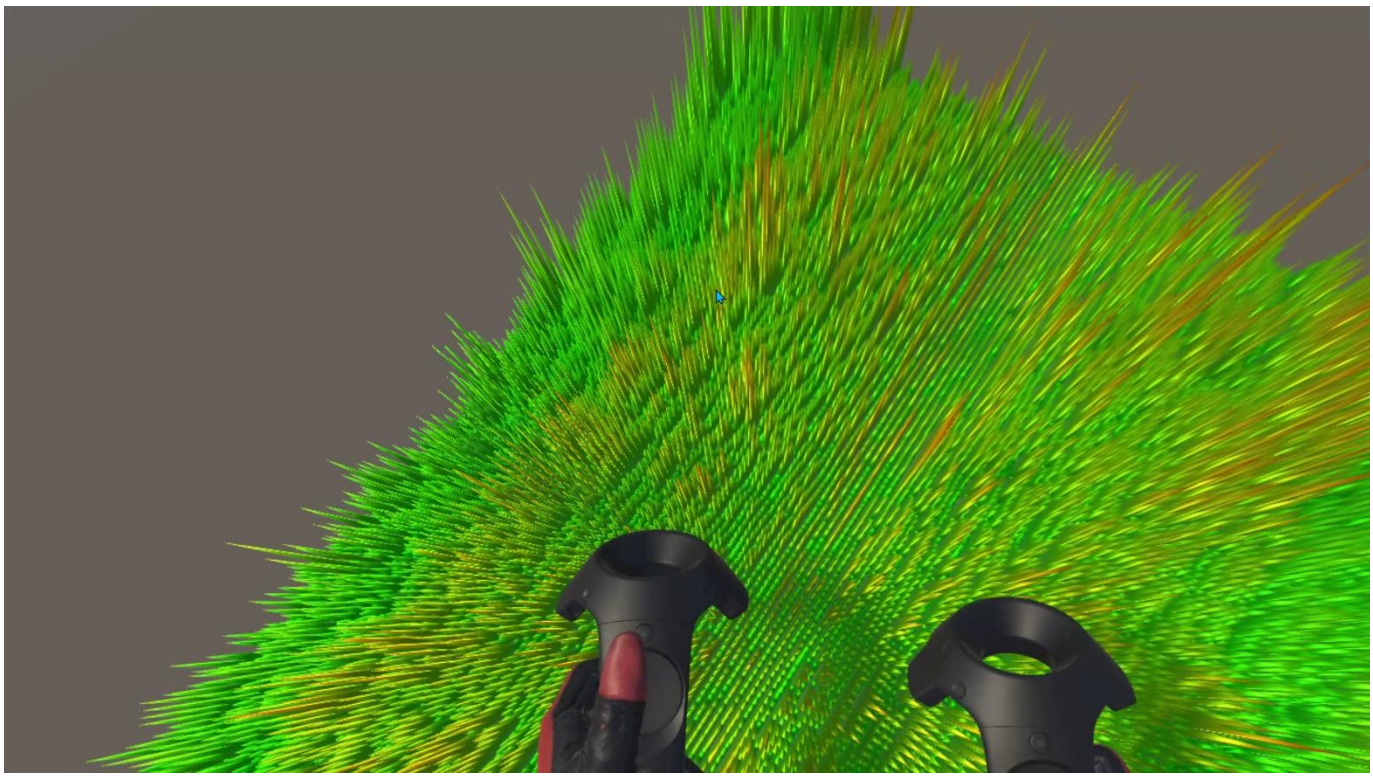

Abb. 3 – Anzeige des Spektrums nach Laden der Audio-Datei (Detailansicht)

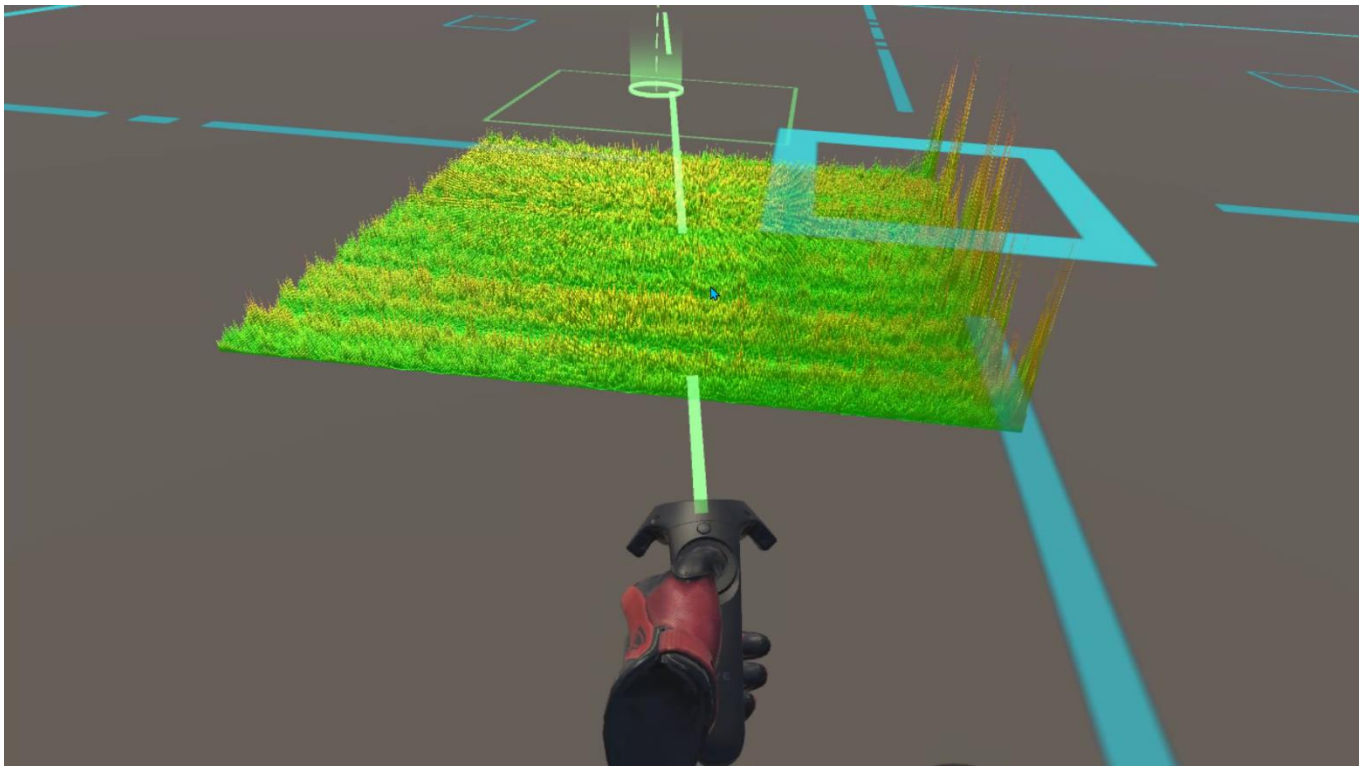

Abb. 4 – Teleportation

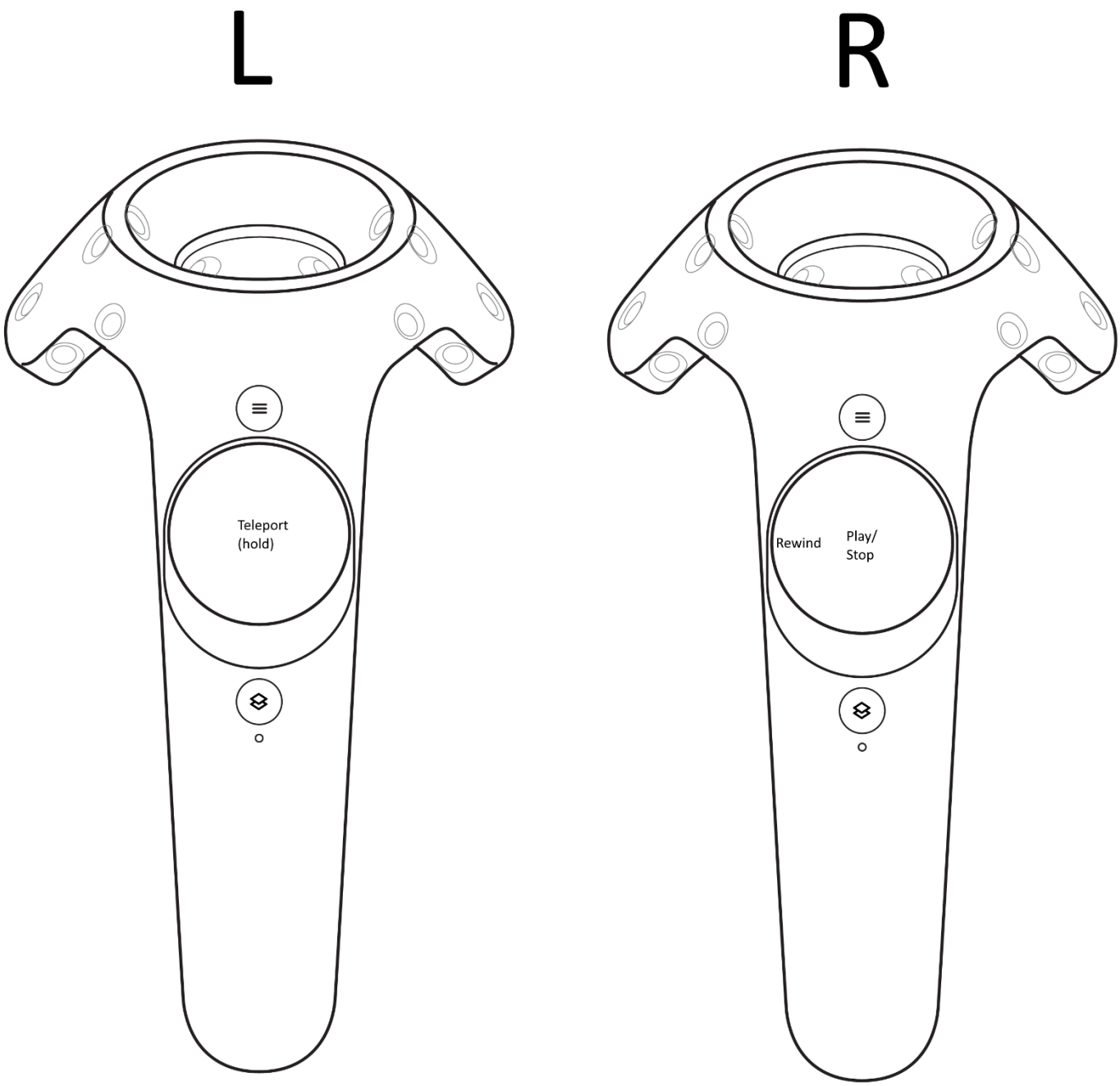

Abb. 5 – Belegung der htc VIVE Controller

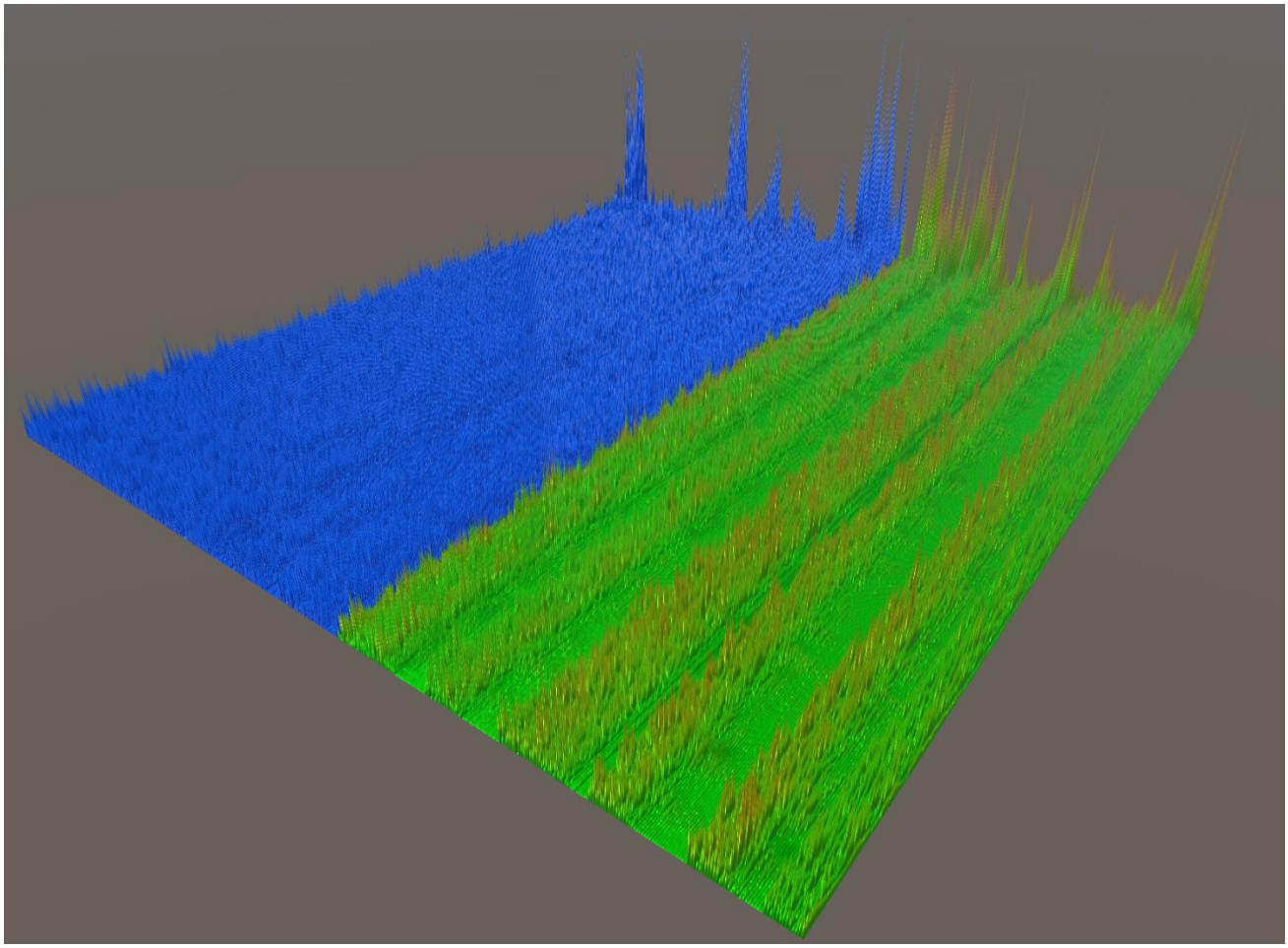

Abb. 6 – Anzeige der Wiedergabeposition

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
▼ PlayerLoop	90.0%	0.2%	2	0.7 KB	15.19	0.03
▼ Initialization.PlayerUpdateTime	77.2%	0.0%	1	0 B	13.03	0.00
WaitForTargetFPS	77.2%	77.2%	1	0 B	13.02	13.02
▶ Camera.Render	11.2%	0.1%	1	0 B	1.89	0.02
▶ GUI.Repaint	0.3%	0.0%	1	0.7 KB	0.05	0.01

Abb. 7 – Unity Profiler – Auslastung der CPU

6 Literatur

Hausstädtler, Uwe, *Der Einsatz von Virtual Reality in der Praxis*, Handbuch für Studenten und Ingenieure, 2., überarbeitete Auflage, Rhombos Verlager, 2010, Berlin, ISBN 978-3-941216-14-3

Jerald, Jason, NextGen Interactions. **–** *The VR Book – Human-Centered Design for Virtual Reality*, A publication in the ACM Books series, #8, First Edition, Association for Computing Machinery and Morgan & Claypool, 2016, San Rafael, ISBN 978-1-97000-112-9

Schäfer, Dmitrij – *Untersuchung der Bewegungsmethoden im virtuellen Raum mit Bezug auf Cybersickness*, Bachelorthesis eingereicht im Rahmen der Bachelorprüfung im Studiengang Angewandte Informatik am Department Informatik der Fakultät Technik und Informatik der Hochschule für Angewandte Wissenschaften Hamburg, 2017, Hamburg

https://users.informatik.haw-hamburg.de/~ubicomp/arbeiten/bachelor/d.schaefer.pdf (letzter Abruf: 28.02.2020)

7 Sonstige Quellen

FFTW 3.3.8 Documentation

http://www.fftw.org/fftw3_doc/ (letzter Abruf: 01.11.2019)

Github, FFTWSharp, Readme & Example code, Tamas Zalay

https://github.com/tszalay/FFTWSharp (letzter Abruf: 05.10.2019)

htc VIVE, Wie sind die Systemanforderungen?

https://www.vive.com/de/support/vive/category_howto/what-are-the-system-requirements.html (letzter Abruf: 04.02.2020)

NAudio, Audio and MIDI library for .NET, Readme & Documentation

https://github.com/naudio/NAudio (letzter Abruf: 04.02.2020)

Profiling Applications Made with Unity, Copyright © 2020 Unity Technologies

https://learn.unity.com/tutorial/profiling-applications-made-with-unity# letzter Abruf: 03.03.2020)

Unity Forums: Performance implications between vertex colors vs textures, bgolus

https://forum.unity.com/threads/performance-implications-between-vertex-colors-vs-textures.671698/#post-4496353 (letzter Abruf: 04.03.2020)

Unity User Manual, Copyright © 2020 Unity Technologies. Publication: 2019.3-003O.

https://docs.unity3d.com/Manual/index.html (letzter Abruf: 20.02.2020)

YouTube: MESH GENERATION in Unity – Basics, Brackeys

https://www.youtube.com/watch?v=eJEpeUH1EMg (letzter Abruf: 05.10.2019)

8 Anhänge

AudioEngine.cs

```
using System.Collections;
using System.Collections.Generic;
using NAudio;
using SimpleFileBrowser;
using UnityEngine;

public class AudioEngine : MonoBehaviour
{
    // just for testing in Unity play mode
    public enum testFiles
    {
        sinesweep1HzTo48000Hz,
        sine4000Hz,
        sine100Hz,
        test
    }
    public testFiles selectAudioTestFile;

    public string filePath;

    [HideInInspector]
    public double[][] fftData;

    private NAudio.Wave.WaveFileReader waveReader;
    private NAudio.Wave.WaveOut waveOut;
    private LoopStream loopStream;

    private double[] importDataAsSamples;
    public int importSampleRate;
    public int importBitDepth;
    public double importDurationInMs;
    public double[] fftFrequencies;
    public int fftBinCount;

    public bool isPlaying;

    private SpectrumMeshGenerator spectrum;

    // Make sure audio engine is loaded before Start() routines of other GameObjects
    void Awake()
    {

        this.spectrum = GameObject.Find("SpectrumMesh").GetComponent<SpectrumMeshGenerator>();

        if (!Application.isEditor)
        {
            // Show file dialog when in standalone mode
            FileBrowser.SetFilters(true, new FileBrowser.Filter("Audio", ".wav", ".aiff", ".mp3",
".m4a", ".ogg"));
            FileBrowser.AddQuickLink("Examples", Application.dataPath + "/Resources/Audio/", null);
            StartCoroutine(ShowLoadDialogCoroutine());
        } else {

            // Load audio data based on selection in AudioEngine component in Unity editor mode
            switch (this.selectAudioTestFile)
            {
                case testFiles.sinesweep1HzTo48000Hz:
                    this.filePath = Application.dataPath + "/Resources/Audio/" +
"sinesweep_1Hz_48000Hz_-3dBFS_30s.wav";
                    break;

                case testFiles.sine4000Hz:
                    this.filePath = Application.dataPath + "/Resources/Audio/" + "sinus4000hz-
10db.wav";
                    break;

                case testFiles.sine100Hz:
```

```
                    this.filePath = Application.dataPath + "/Resources/Audio/" + "sinus100hz-
10db.wav";
                    break;

                case testFiles.test:
                    this.filePath = Application.dataPath + "/Resources/Audio/" + "test.wav";
                    break;
            }
            this.LoadAudioData();
        }
    }

    IEnumerator ShowLoadDialogCoroutine()
    {
        // Show a load file dialog and wait for a response from user
        yield return FileBrowser.WaitForLoadDialog(false, null, "Load audio file", "Load");

        Debug.Log(FileBrowser.Success + " " + FileBrowser.Result);

        if (FileBrowser.Success)
        {
            this.filePath = FileBrowser.Result;
            this.LoadAudioData();
        }
    }

    public void LoadAudioData()
    {
        // Read in wav file and convert into an array of samples
        this.waveReader = new NAudio.Wave.WaveFileReader(this.filePath);
        int numOfBytes = (int)this.waveReader.Length;
        this.importSampleRate = this.waveReader.WaveFormat.SampleRate;
        this.importBitDepth = this.waveReader.WaveFormat.BitsPerSample;
        this.importDurationInMs = this.waveReader.TotalTime.TotalMilliseconds;

        byte[] importDataAsBytes = new byte[numOfBytes];
        this.waveReader.Read(importDataAsBytes, 0, numOfBytes);


        // Convert into double array
        // 8, 16, 24, 32 and 64 bits per sample are supported for now
        int numOfSamples = numOfBytes / (this.importBitDepth / 8);
        this.importDataAsSamples = new double[numOfSamples];

        switch (this.importBitDepth)
        {
            case 8:
                for (int i = 0; i < numOfSamples; i++)
                {
                    this.importDataAsSamples[i] = importDataAsBytes[i];
                }
                break;
            case 16:
                for (int i = 0; i < numOfSamples; i++)
                {
                    this.importDataAsSamples[i] =
(double)System.BitConverter.ToInt16(importDataAsBytes, i);
                }
                break;
            case 24:
                for (int i = 0; i < numOfSamples - 2; i++)
                {
                    this.importDataAsSamples[i] = (double)(importDataAsBytes[i] + (importDataAsBytes[i
+ 1] << 8) + (importDataAsBytes[i + 2]) << 16);
                }
                break;
            case 32:
                for (int i = 0; i < numOfSamples; i++)
                {
                    this.importDataAsSamples[i] =
(double)System.BitConverter.ToInt32(importDataAsBytes, i);
                }
                break;
            case 64:
```

```
                for (int i = 0; i < numOfSamples; i++)
                {
                    this.importDataAsSamples[i] =
(double)System.BitConverter.ToInt64(importDataAsBytes, i);
                }
                break;
            default:
                //TODO do some error handling here
                break;
        }

        this.DoFft();
        this.spectrum.Init();
    }

    public void Play()
    {
        Debug.Log("<AudioEngine> Play");
        if (this.waveOut == null)
        {
            this.loopStream = new LoopStream(this.waveReader);
            this.waveOut = new NAudio.Wave.WaveOut(); //TODO: find fix. Waveout will try to use
Windows.Forms for error dialog boxes - this is not supported under Unity Mono (will display "could not
register the window class, win 32 error 0")
            this.waveOut.PlaybackStopped += OnPlaybackStopped;
            this.waveOut.Init(this.loopStream);
            this.waveOut.Play();
        } else {
            this.waveOut.Play();
        }
        this.isPlaying = true;
    }

    public void Stop()
    {
        this.waveOut.Stop();
        this.isPlaying = false;
        this.spectrum.ResetMeshColors();

        Debug.Log("<AudioEngine> Stop");
    }

    private void OnPlaybackStopped(object sender, System.EventArgs e)
    {
        Debug.Log("<AudioEngine> " +e.ToString());
    }

    public double GetPositionInMs()
    {
        long bytePos = this.loopStream.Position;
        double ms = bytePos * 1000.0 / this.importBitDepth / 1 * 8 / this.importSampleRate; //1 for
mono, TODO multichannel

        return ms;
    }

    public void SetAudioLooping(bool loop)
    {
        this.loopStream.EnableLooping = loop;
    }

    public void Rewind()
    {
        Debug.Log("<AudioEngine> Rewind");
        if (this.waveOut != null)
        {
            this.waveReader.Position = 0;
            this.spectrum.ResetMeshColors();
        }
    }

    public void StopAudioEngine()
    {
        if (this.waveOut != null)
```

```
        {
            this.waveOut.Dispose();
            this.waveOut = null;
        }

        if (this.waveReader != null)
        {
            this.waveReader.Close();
            this.waveReader.Dispose();
            this.waveReader = null;
        }

        this.isPlaying = false;
    }


    public void DoFft()
    {
        if (this.importDataAsSamples != null && this.importDataAsSamples.Length > 0)
        {
            int fftSize = (int)(this.importSampleRate * 0.0232199546485261); // magic number taken
from 44.100 Hz / 1024
            //int fftSize = 2048;
            this.fftBinCount = fftSize / 2;

            // Calculate size of chunk that will be sent to FFT routine
            //int chunkSize = this.importSampleRate / 2 / 100; // 50ms
            //int chunkSize = this.importSampleRate / 2; // 500ms
            int chunkSize = fftSize;
            int numOfChunks = (this.importDataAsSamples.Length + chunkSize - 1) / chunkSize; //
integer round up

            Debug.Log("<AudioEngine> numOfSamples: " + this.importDataAsSamples.Length.ToString() + "
chunkSize: " + chunkSize.ToString() + " numOfChunks: " + numOfChunks.ToString() + " binResolution: " +
(this.importSampleRate / fftSize).ToString() + "Hz");

            // Create map of frequencies for the bins
            this.fftFrequencies = new double[this.fftBinCount];
            for (int i = 0; i < this.fftBinCount; i++)
                this.fftFrequencies[i] = (double)i / this.fftBinCount * this.importSampleRate / 2;

            // Do FFT per chunk
            double[][] input = new double[numOfChunks][];
            double[][] result = new double[numOfChunks][];

            Fft fft = new Fft(fftSize, Fft.WindowType.hamming, this.importSampleRate);

            for (int i = 0; i < numOfChunks; i++)
            {
                input[i] = new double[chunkSize];

                // Due to aliasing the second part would be redundant, so the ouput array is fftSize /
2 + 1
                // as there is no need to store mirrored information
                result[i] = new double[fftSize / 2 + 1];

                for (int j = 0; j < chunkSize; j++)
                {
                    // last chunk might be smaller than chunkSize or chunkSize smaller than fft size
                    if (this.importDataAsSamples.Length > i * chunkSize + j)
                    {
                        input[i][j] = this.importDataAsSamples[i * chunkSize + j];
                    } else {
                        // fill with zeros
                        input[i][j] = 0.0;
                    }
                }
                fft.Run(input[i], result[i]);
            }
            this.fftData = result;
            fft.Dispose();
        }
    }
```

```
    private void Update()
    {
        if (Input.GetButtonDown("PlayStop"))
        {
            if (!this.isPlaying)
            {
                this.Play();
            } else {
                this.Stop();
            }
        }

        if (Input.GetButtonDown("Rewind"))
        {
            this.Rewind();
        }
    }

    void OnApplicationQuit()
    {
        Debug.Log("Exit");
        this.StopAudioEngine();
    }
}
```

Fft.cs

```
using System.Collections;
using System.Collections.Generic;
using System.Runtime.InteropServices;
using FFTWSharp;

// FFT uses FFTW - see http://www.fftw.org/index.html
// C# Wrapper based on the documentation of FFTWSharp - see https://github.com/tszalay/FFTWSharp

public class Fft
{
    public enum WindowType
    {
        flat,
        hann,
        hamming,
        blackman
    };
    private WindowType winfunc;
    private int fftLength;
    private double[] fftWindow, dout, din;
    private System.IntPtr pin, pout, fplan;
    public double ampCf, pwrCf;

    public Fft(int n, WindowType wt, int sampleRate)
    {
        this.fftLength = n;
        this.pin = fftw.malloc(n * 2 * sizeof(double));
        this.pout = fftw.malloc(n * 2 * sizeof(double));

        this.din = new double[n];
        this.dout = new double[n/2+1];
        this.fftWindow = new double[n];

        this.winfunc = wt;
        this.MakeFftWindow();

        for (int i = 0; i < n; i++)
            this.din[i] = 0.0;

        this.fplan = fftw.dft_1d(n, this.pin, this.pout, fftw_direction.Forward, fftw_flags.Measure);
    }

    private void MakeFftWindow()
    {
        double alpha, a0, a1, a2;

        switch (this.winfunc)
        {
            case WindowType.hann:
                for (int i = 0; i < this.fftLength; i++)
                    fftWindow[i] = 0.5 - 0.5 * System.Math.Cos((double)i * 2 * System.Math.PI /
(this.fftLength - 1));
                this.ampCf = 6;
                this.pwrCf = 4.3;
                break;

            case WindowType.hamming:
                for (int i = 0; i < this.fftLength; i++)
                    fftWindow[i] = 0.54 - 0.46 * System.Math.Cos((double)i * 2 * System.Math.PI /
(this.fftLength - 1));
                this.ampCf = 5.35;
                this.pwrCf = 4.0;
                break;

            case WindowType.blackman:
                alpha = 0.16;
                a0 = (1.0 - alpha) / 2.0;
                a1 = 0.5;
                a2 = alpha / 2;
                for (int i = 0; i < this.fftLength; i++)
                    fftWindow[i] =
```

```
                        a0
                        - a1 * System.Math.Cos((double)i * 2 * System.Math.PI / (this.fftLength - 1))
                        + a2 * System.Math.Cos((double)i * 4 * System.Math.PI / (this.fftLength - 1));
                this.ampCf = 7.54;
                this.pwrCf = 5.2;
                break;

            case WindowType.flat:
            default:
                for (int i = 0; i < this.fftLength; i++)
                    fftWindow[i] = 1.0;
                this.ampCf = 0;
                this.pwrCf = 0;
                break;

        }

    }

    private void DoFft(double[] outp)
    {
        double c = this.fftLength;
        double cf = 2.0 / 32767.0;

        Marshal.Copy(this.din, 0, this.pin, this.fftLength);
        fftwf.execute(this.fplan);

        Marshal.Copy(this.pout, this.dout, 0, this.fftLength / 2 + 1);
        for (int i = 0; i < this.fftLength / 2; i++)
            //outp[i] = cf * System.Math.Sqrt((this.dout[2 * i] * this.dout[2 * i] + this.dout[2 * i +
1] * this.dout[2 * i + 1]) / c);
            outp[i] = cf * System.Math.Sqrt((this.dout[i] * this.dout[i] + this.dout[i + 1] *
this.dout[i + 1]) / c);


    }

    public void Run(double[] inp, double[] outp)
    {
        for (int i = 0; i < this.fftLength; i++)
            this.din[i] = inp[i] * this.fftWindow[i];
        this.DoFft(outp);
    }
    public void Dispose()
    {
        fftwf.free(this.pin);
        fftwf.free(this.pout);
        fftwf.destroy_plan(this.fplan);
    }
}
```

SpectrumMeshGenerator.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class SpectrumMeshGenerator : MonoBehaviour
{
    public float dataScale;
    public float edgeLengthOfRaster;

    private AudioEngine audioEngine;

    private GameObject[] meshObj;
    private MeshFilter[] mFilters;
    private MeshRenderer[] mRenderers;
    private Mesh[] meshes;
    private Vector3[][] vertices;
    private int[][] triangles;
    private Color32[][] colors;

    private int countOfRasterVertices;
    private int countOfPeakVertices;

    private void Update()
    {
        if (Input.GetButtonDown("ScaleMeshYDec"))
        {
            this.ScaleMeshY(-0.05f);
        }

        if (Input.GetButtonDown("ScaleMeshYInc"))
        {
            this.ScaleMeshY(0.05f);
        }

        // Color meshes according to play position
        if (this.audioEngine.isPlaying)
        {
            double msPerChunk = this.audioEngine.importDurationInMs / this.audioEngine.fftData.Length;
            int posIdx = (int)(this.audioEngine.GetPositionInMs() / msPerChunk);

            for (int i = 0; i < posIdx; i++)
            {
                this.mRenderers[i].material = Resources.Load("Materials/SpectrumMatPlaying") as
Material;
            }

            // workaround until loopstream event will work to trigger reset of spectrum material
            for (int i = posIdx; i < this.audioEngine.fftData.Length; i++)
            {
                this.mRenderers[i].material = Resources.Load("Materials/SpectrumMat") as Material;
            }
        }
    }

    private void Start()
    {

    }


    public void Init()

    {
        this.audioEngine = GameObject.Find("Audio").GetComponent<AudioEngine>();

        this.meshObj = new GameObject[this.audioEngine.fftData.Length];
        this.mFilters = new MeshFilter[this.audioEngine.fftData.Length];
        this.mRenderers = new MeshRenderer[this.audioEngine.fftData.Length];

        this.meshes = new Mesh[this.audioEngine.fftData.Length];
        this.vertices = new Vector3[this.audioEngine.fftData.Length][];
```

```
        this.triangles = new int[this.audioEngine.fftData.Length][];
        this.colors = new Color32[this.audioEngine.fftData.Length][];

        for (int i=0; i < this.audioEngine.fftData.Length; i++)
        {
            // Add GOs, MFs and MRs
            this.meshObj[i] = new GameObject("spectrumMesh" + i.ToString());
            this.meshObj[i].transform.parent = this.transform;

            this.mFilters[i] = meshObj[i].AddComponent<MeshFilter>();
            this.mFilters[i].name = "FFTData" + i.ToString();

            this.mRenderers[i] = meshObj[i].AddComponent<MeshRenderer>();
            this.mRenderers[i].material = Resources.Load("Materials/SpectrumMat") as Material;

            this.meshes[i] = new Mesh();
            //this.meshes[i].indexFormat = UnityEngine.Rendering.IndexFormat.UInt32; // Increase from
16bit as this would only allow 65.536 vertices per mesh
            this.meshes[i].Clear();
            this.mFilters[i].mesh = this.meshes[i];

            // Create Spectrum polygons
            this.SetVertices(i);
            this.SetMeshColors(i);
            this.SetTriangles(i);
            this.meshes[i].RecalculateBounds();
            this.meshes[i].RecalculateNormals();
            this.meshes[i].Optimize();

            /*
            // Text for frequency legend - very crude code for testing
            TextMesh textMesh = GameObject.Find("FreqLegend").GetComponent<TextMesh>();
            string frequencies = "";

            for (int f = this.audioEngine.fftFrequencies.Length - 1; f >= 0; f--)
            {
                frequencies += System.Math.Round(this.audioEngine.fftFrequencies[f], 2).ToString() + "
Hz\n";
            }
            textMesh.text = frequencies;
            */
        }


        /*
        // Testing output

        for (int i=0; i < this.meshes[0].vertices.GetLength(0); i++)
        {
            Debug.Log("<SpectrumMesh> " + this.meshes[0].vertices[i].x.ToString() + " " +
this.meshes[0].vertices[i].y.ToString() + " " + this.meshes[0].vertices[i].z.ToString());
        }
        */
    }

    /*
    // Visualize vertices for Scene view debugging
    // (huge performance killer for lots of vertices, will freeze unity if used for the whole fft
data)
    private void OnDrawGizmos()
    {
        if (this.audioEngine == null)
        {
            return;
        }

        // Draw Spheres for vertices
        var transform = this.transform;
        foreach (var vert in this.meshes[0].vertices)
        {
            Gizmos.color = Color.red;
            Gizmos.DrawSphere(transform.TransformPoint(vert), 0.003f);
        }
```

```
    }*/

    private void SetVertices(int meshIdx)
    {
        this.countOfRasterVertices = 2 * (this.audioEngine.fftBinCount + 1);
        this.countOfPeakVertices = this.audioEngine.fftBinCount;

        this.vertices[meshIdx] = new Vector3[this.countOfRasterVertices + this.countOfPeakVertices];

        Vector3 center = transform.parent.position;

        // Add vertices
        // first the ground raster vertices, then all fft value peak vertices

        // Add ground raster vertices
        for (int i = 0; i < this.countOfRasterVertices / 2; i++)
        {
            Vector3 r1;
            r1.x = center.x + i * this.edgeLengthOfRaster;
            r1.y = center.y;
            r1.z = center.z + meshIdx * this.edgeLengthOfRaster;

            Vector3 r2;
            r2.x = center.x + i * this.edgeLengthOfRaster;
            r2.y = center.y;
            r2.z = center.z + meshIdx * this.edgeLengthOfRaster + this.edgeLengthOfRaster;

            this.vertices[meshIdx][i] = r1;
            this.vertices[meshIdx][i + this.audioEngine.fftBinCount + 1] = r2;
        }

        // Add peak vertices
        for (int i = 0; i < this.countOfPeakVertices; i++) {
            Vector3 p;
            p.x = center.x + i * this.edgeLengthOfRaster + this.edgeLengthOfRaster / 2;
            p.y = center.y + (float)this.audioEngine.fftData[meshIdx][i] * this.dataScale;
            p.z = center.z + meshIdx * this.edgeLengthOfRaster + this.edgeLengthOfRaster / 2;

            this.vertices[meshIdx][this.countOfRasterVertices+i] = p;
        }
        this.meshes[meshIdx].vertices = this.vertices[meshIdx];
    }

    public void ResetMeshColors()
    {
        for (int i = 0; i < this.audioEngine.fftData.Length; i++)
        {
            this.mRenderers[i].material = Resources.Load("Materials/SpectrumMat") as Material;
        }
    }

    private void SetMeshColors(int meshIdx)
    {
        this.colors[meshIdx] = new Color32[this.vertices[meshIdx].Length];

        float max = 0;
        for (int i = 0; i < this.vertices[meshIdx].Length; i++)
        {
            if (this.vertices[meshIdx][i].y > max)
                max = this.vertices[meshIdx][i].y;
        }

        for (int i = 0; i < this.vertices[meshIdx].Length; i++)
            this.colors[meshIdx][i] = Color.Lerp(Color.green, Color.red, this.vertices[meshIdx][i].y /
max);

        this.meshes[meshIdx].colors32 = this.colors[meshIdx];
    }

    private void SetTriangles(int meshIdx)
    {
        this.triangles[meshIdx] = new int[this.audioEngine.fftBinCount * 4 * 3]; //4 pyramid sides per
fft bin (without base), 3 vertices indices per side
```

```
        for (int i = 0; i < this.triangles[meshIdx].Length - 12; i += 12)
        {
            this.triangles[meshIdx][i] = this.countOfRasterVertices + i / 12;
            this.triangles[meshIdx][i + 1] = i / 12;
            this.triangles[meshIdx][i + 2] = this.audioEngine.fftBinCount + 1 + i / 12;

            this.triangles[meshIdx][i + 3] = this.countOfRasterVertices + i / 12;
            this.triangles[meshIdx][i + 4] = this.audioEngine.fftBinCount + 2 + i / 12;
            this.triangles[meshIdx][i + 5] = 1 + i / 12;

            this.triangles[meshIdx][i + 6] = this.countOfRasterVertices + i / 12;
            this.triangles[meshIdx][i + 7] = this.audioEngine.fftBinCount + 1 + i / 12;
            this.triangles[meshIdx][i + 8] = this.audioEngine.fftBinCount + 2 + i / 12;
            this.triangles[meshIdx][i + 9] = this.countOfRasterVertices + i / 12;
            this.triangles[meshIdx][i + 10] = 1 + i / 12;
            this.triangles[meshIdx][i + 11] = i / 12 ;

        }
        this.meshes[meshIdx].triangles = this.triangles[meshIdx];
    }

    public void ScaleMeshY(float offset)
    {
        Vector3 scale = gameObject.transform.localScale;
        Debug.Log(scale.ToString());
        gameObject.transform.localScale.Set(scale.x, scale.y + offset, scale.z);
    }
}
```