VST-Implementierung Ambisonischer Nachhalleffekte

TI-Projekt

Sebastian Grill

Betreuung: Dr. Franz Zotter

Graz, 7. Dezember 2017

kunst uni graz

institut für elektronische musik und akustik

Zusammenfassung

Am Institut für Elektronische Musik und Akustik (IEM) wurden im Rahmen einer Konzertveranstaltung einige Halleffekte mit dreidimensionaler Wiedergabe auf Basis von Ambisonics entwickelt. Dabei wurden Rückkopplungsnetzwerke mit diversen Verzögerungen und eine Misch-/Rotationsmatrix verwendet, die im Sequenzer implementiert waren. Im Rahmen der Projektarbeit sollen diese Effekte in Form von VST-Plugins effizient implementiert werden, um sie BenutzerInnen außerhalb des IEM leichter zugänglich machen zu können. Weiters soll die Arbeit durch ihre Dokumentation eine Grundlage schaffen, um für etwaiige zukünftige Arbeiten den Einstieg in das Erstellen von VST-Plugins sowie die Verwendung der dazu notwendige Toolchain zu erleichtern.

Abstract

The Institute of Electronic Music and Acoustics (IEM) has, in the course of a concert, developed several spatial reverberation effects based on Ambisonics. To this end, feedback loops with varying delay as well as a mixing/rotation matrix were implemented in a sequencer. The aim of this student project is to efficiently implement these algorithms as VST plugins so as to make them easily employable by users outside of the IEM. Furthermore, this work with its documentation should form a basis for eventual future projects and provide an introduction to VST plugin creation and the associated toolchain.

Inhaltsverzeichnis

1 Einleitung

Ambisonics ist aktuell ein sehr aktiver Zweig der Forschung am Institut für Elektronische Musik und Akustik (IEM). Ambisonischen Aufnahmen beziehungsweise Produktionen ist zueigen, dass Effekte durch die räumliche Wiedergabe zusätzlich zu den bei allen Wiedergabeverfahren üblichen und etablierten Optionen der Beinflussung des Klanges bzw. Höreindruckes auch eine vergleichsweise einfache und gleichzeitig uneingeschränkte Manipulation der Richtungsinformation einer Schallquelle vornehmen können.

Abbildung 1 – Sphärisches Mikrophonarray des IEM

Eine Möglichkeit dies auszunutzen ist die Erstellung von für den/die HörerIn besonders immersiven Halleffekten. State of the Art bei konventionellen Halleffekten ist die Faltung des Audiosignals mit der zuvor gemessenen Impulsantwort eines Raumes, dessen Verhalten nachgebildet werden soll. In Kombination mit Ambisonics bringt dieses Verfahren auch Anforderungen und Einschränkungen mit sich. Die Faltung ist vor allem bei langen Impulsantworten ausgesprochen rechenintensiv. Erwägt man dabei auch noch die große Kanalanzahl von Ambisonics hoher Ordnung, stößt man unter Umständen selbst mit modernen, leistungsfähigen Rechnern im Livebetrieb an die Grenze des Machbaren. Als praktische Herausforderung gilt auch, dass die verwendeten Raumimpulsantworten eine geeignete Richtungsauflösung besitzen müssen, um einen sinnvollen Einsatz zu recht-

fertigen. Weil ein reichhaltiger Satz an gemessenen Hallkonfigurationen die wesentliche Anpassungsmöglichkeit ist, kann ein Faltungshall nicht in jedem Fall die praktischen Bedürfnisse befriedigen.

Rückkopplungsnetze können der physikalischen Modellierung von Diffusfeldern zugerechnet werden und sind eine äußerst effektive Alternative. Im Rahmen dieser Arbeit wurde ein solches Rückkopplungsnetz für die Verwendung mit Ambisonics als VST-Plugin mit graphischer Benutzeroberfläche implementiert und die Erfahrung des IEM aus anderen Implementierungen berücksichtigt. Für den hier entwickelten Halleffekt ist eine Einbindung in eine Ambisonics Effektlibrary des IEM vorgesehen, die unter einer Open Source Lizenz veröffentlicht werden soll.

2 Rückkopplungsnetze als Nachhalleffekt

2.1 Herkömmliches Rückkopplungsnetz

Die Standardform eines Rückkopplungsnetzes ist gegeben als

$$y(n) = \mathbf{c}^T \mathbf{s}(n) + dx(n) \tag{1}$$

$$\mathbf{s}(n + \mathbf{m}) = \mathbf{A}\mathbf{s}(n) + \mathbf{b}x(n) \tag{2}$$

$x(n)$ und $y(n)$ sind die zeitlich veränderlichen Ein- sowie Ausgangssignale darstellen. $\mathbf{A} \in \mathbb{C}^{N \times N}$ bezeichnet die Rückkopplungsmatrix, $\mathbf{b}$ und $\mathbf{c}$ sind $N \times 1$ Vektoren mit der Eingangs- und Ausgangsverstärkung, und d bezeichnet den Durchgriffsterm des Direktsignals. $\mathbf{m}$ ist ein $N \times 1$ Vektor mit den jeweiligen Längen der Verzögerungskomponenten. N bezeichnet die Ordnung des Netzes. [SH15]

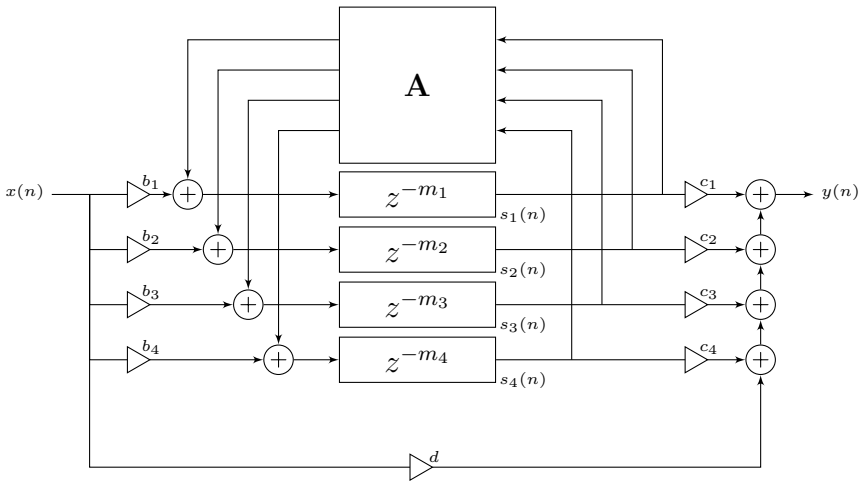

Abbildung 2 – Standardform eines Rückkopplungsnetzes zur Verwendung als Nachhall (nach [SP82])

2.2 Rückkopplungsnetz für Systeme mit mehrkanaligem Ein- und Ausgang

Mit kleinen Modifikationen ist es möglich, ein solches Netzwerk für die Verhallung von beispielsweise ambisonischen Signalen zu adaptieren (siehe [Blo17], beruhend auf einer ursprünglichen Umsetzung von Daniel Rudrich).

Die Beschreibung des Systems wird dann zu

$$\mathbf{y}(n) = \mathbf{c} \circ \mathbf{s}(n) + \mathbf{d} \circ \mathbf{x}(n) \tag{3}$$

$$\mathbf{s}(n + \mathbf{m}) = \mathbf{A}\mathbf{s}(n) + \mathbf{b} \circ \mathbf{x}(n) \tag{4}$$

wobei der Operator ∘ das Hadamard-Produkt (elementweise Multiplikation) beschreibt.

Eine für diese Arbeit entscheidende Erkenntnis aus [Blo17] ist, dass der für Rückkopplungsnetze niedriger Ordnung charakteristische „blecherne“ Klang bei Netzen höherer Ordnungen vollständig verschwindet. Daher kann bei ausreichend großer Ambisonischer Ordnung K, die quadratisch in die Ordnung $N = (K+1)^2$ des Rückkopplungsnetzes eingeht, auf die Maßnahme einer zeitlichen Modulation in der Rückkopplungsmatrix verzichtet werden.

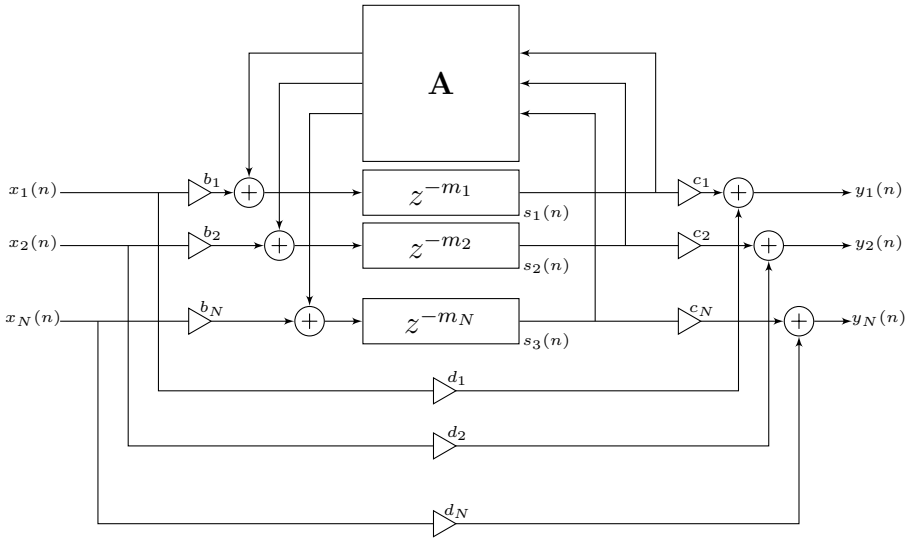

Abbildung 3 – Adaptierte Form zur Verwendung mit mehrfachem Ein- Ausgangssignal

2.3 Abschätzung der Nachhallzeit

Eine Methode zur Abschätzung der Nachhallzeit eines Rückkopplungsnetzes wird in [SH17] beschrieben.

Um die Verluste in der Rückkopplung und damit das zeitliche Ausklingverhalten exakt zu steuern, wir die Mischmatrix als kompliziertester Teil des Rückkopplungsnetzes unitär und damit verlustlos gemacht, was ein Ausklingen völlig verhindert und grenzstabil ist. Für eine reellwertige Rechnung genügt, dass die Rückkopplungsmatrix $\mathbf{A}$ orthogonal ist

$$\mathbf{A}\mathbf{A}^T = \mathbf{I}. \tag{5}$$

Entwurfsmöglichkeiten für $\mathbf{A}$ werden in Abschnitt 3.1.5 besprochen. Die Verluste und damit das Ausklingverhalten werden über die Gewichte $\mathbf{g}$ gesteuert (siehe Abb. 5) und für die gewünschte Nachhallzeit (in diesem Fall T_{60}), ergibt sich sich die Dämpfung pro Abtastintervall

$$\delta(\omega) = -\frac{60}{T_{60}(\omega) f_s} \text{ in } dB \text{ [SH17]}. \tag{6}$$

Da jeder Abtastwert pro Durchlauf des Netzwerkes m_i Abtastintervalle verzögert wird, muss die Dämpfung pro Kanal mit dem Produkt der jeweiligen Delaylänge m_i und der Abtastrate f_s potenziert werden, um einen gleichmäßigen Energieabfall in allen Kanälen zu gewährleisten. Die Dämpfung für den Kanal i mit Verzögerung m_i ist demnach

$$g_i(\omega) = -\frac{60 m_i}{T_{60}(\omega)} \text{ in } dB. \tag{7}$$

2.4 Die schnelle Walsh-Hadamard-Transformation

Gemäß Gleichung 4, die das Rückkopplungssystem beschreibt, muss für jeden abgetasteten Zeitpunkt n die Multiplikation $\mathbf{A}\mathbf{s}(n)$ ausgeführt werden. Die Komplexität dieser Matrix-Vektor-Multiplikation liegt in der Größenordnung $O(N^2)$. Selbst unter Zuhilfenahme von moderner Hardware und Parallelisierungsverfahren (Single Instruction Multiple Data - SIMD) kann der Komplexitätsgrad bei höheren Systemordnungen problematisch werden [1].

Hadamard-Matrizen werden in der Praxis oft für Rückkopplungsnetze herangezogen, da sie orthogonal sind (siehe Abschnitt 2.3) und aufgrund ihrer Struktur die Energie im Netzwerk sehr diffus verteilen.

Die rekursive Definition einer Hadamard-Matrix der Ordnung m lautet

$$\mathbf{H}_m = \frac{1}{\sqrt{2}} \begin{pmatrix} \mathbf{H}_{m-1} & \mathbf{H}_{m-1} \\ \mathbf{H}_{m-1} & -\mathbf{H}_{m-1} \end{pmatrix} \tag{8}$$

wobei $\mathbf{H}_0 = 1$.

Ist $\mathbf{A}$ eine Hadamard-Matrix, ist es möglich, das Produkt $\mathbf{A}\mathbf{s}(n)$ als schnelle Walsh-Hadamard-Transformation (Fast Walsh Hadamard Transform - FWHT) von $\mathbf{s}(n)$ zu implementieren, ein Verfahren, das nach dem „Divide et impera"-Prinzip vorgeht und die Komplexität auf $O\big(N \cdot log(N)\big)$ Additionen und N Multiplikationen reduziert [a].

a. Die Komplexität verringert sich damit bei Ambisonics 7ter Ordnung von $64^2 = 4096$ Gleikommamultiplikationen auf $64 \cdot log(64) \approx 120$ Additionen und Subtraktionen sowie 64 Gleitkommamultiplikationen.

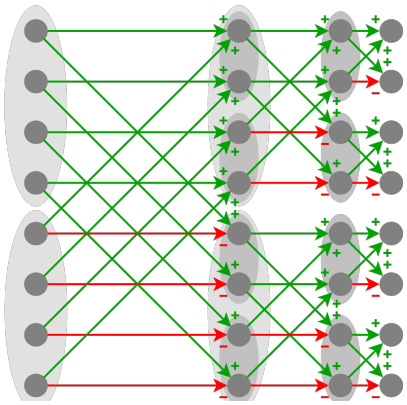

Abbildung 4 – FWHT für einen Vektor der Dimension 8 (Quelle: Wikipedia)

1. Zum Vergleich: Bei Ambisonics 7ter Ordnung liegt die Kanalzahl bei $N = 64$, es müssen also allein für die Operation $\mathbf{A}\mathbf{s}(n)$ $64^2 = 4096$ Gleitkommamultiplikationen pro Abtastzeitpunkt durchgeführt werden.

3 Implementierung und graphische Benutzeroberfläche

3.1 Audiosignalverarbeitung

Abbildung 5 zeigt den Signalfluss der Rückkopplungsstruktur im implementierten VST-Plugin.

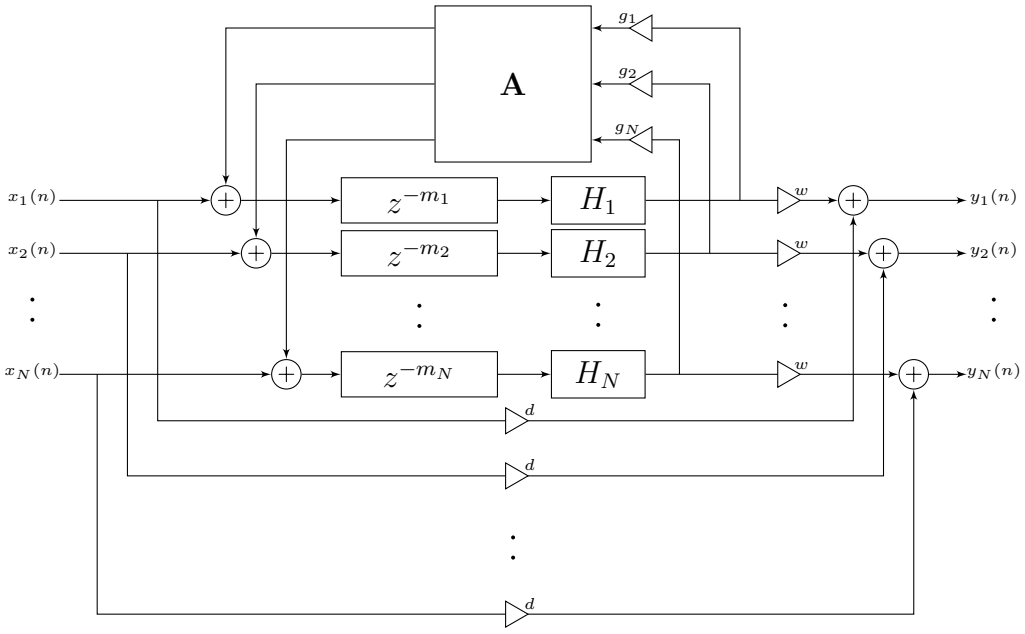

Abbildung 5 – Blockschaltbild des Signalflusses im VST-Plugin

$\mathbf{H}$ bezeichnet eine Filterbank bestehend aus jeweils einem parametrischen Kuschwanz-Tiefpass sowie einem parametrischen Kuhschwanz-Hochpass. Die Filter $H_1 ... H_N$ sind prinzipiell identisch, allerdings ist die Verstärkung eines jeden Filters bei der Grenzfrequenz von der Länge des dazugehörigen Delays abhängig. Die Filter dienen dazu, die Nachhallzeit frequenzabhängig gestalten zu können. Mit $\mathbf{g}$ kann frequenzunabhängig die Absorption pro Reflexion verändert werden. w und d regeln das Verhältnis zwischen verhalltem und direktem Signal.

3.1.1 Rückkopplungsnetzwerk

Die Struktur in Abb. 5 wird von der Funktion *applyFeedbackPath* implementiert:

```
1 inline void FdnReverbAudioProcessor::applyFeedbackPath (AudioBuffer
    <float>& buffer, std::vector<AudioBuffer<float>>&
    delayBufferVector)
2 {
3     const int numSamples = buffer.getNumSamples();
```

```
4
5       // if more channels than network order, mix pairs of high order
            channels
6       // until order == number of channels
7       if (_channelSetting > _networkOrder)
8       {
9           int diff = _channelSetting - _networkOrder;
10          int start_index = _channelSetting - diff * 2;
11
12          for (int num = 0; num < diff; ++num)
13          {
14              int idx = start_index + num;
15              float *const writeIn = buffer.getWritePointer (idx);
16              float *const writeOut1 = buffer.getWritePointer (idx +
                    num);
17              float *const writeOut2 = buffer.getWritePointer (idx +
                    num + 1);
18
19              for (int i = 0; i < numSamples; ++i)
20              {
21                  writeIn[i] = (writeOut1[i] + writeOut2[i]) * 1.f /
                        sqrt(2.f);
22              }
23          }
24      }
25
26      for (int i = 0; i < numSamples; ++i)
27      {
28          // apply delay to each channel for one time sample
29          for (int channel = 0; channel < _networkOrder; ++channel)
30          {
31              const int idx = std::min(channel, _channelSetting);
32              float *const channelData = buffer.getWritePointer (idx)
                    ;
33              float *const delayData = delayBufferVector[channel].
                    getWritePointer (0);
34
35              int delayPos = delayPositionVector[channel];
36
37              // data exchange between IO buffer and delay buffer
38              const float in = channelData[i];
39              if (channel < _channelSetting)
40                  delayData[delayPos] += in;
41
42              // apply shelving filters
43              delayData[delayPos] = highShelfFilters[channel].
                    processSingleSampleRaw(delayData[delayPos]);
44              delayData[delayPos] = lowShelfFilters[channel].
                    processSingleSampleRaw(delayData[delayPos]);
45
46              if (channel < _channelSetting)
47              {
48                  channelData[i] = delayData[delayPos] * delayGain;
49                  channelData[i] += in * directGain;
```

```
50              }
51
52              transferVector[channel] = delayData[delayPos] *
                    feedbackGainVector [channel];
53          }
54
55          // perform fast walsh hadamard transform
56          fwht (transferVector);
57
58          // write back into delay buffer
59          // increment the delay buffer pointer
60          for (int channel = 0; channel < _networkOrder; ++channel)
61          {
62              float *const delayData = delayBufferVector[channel].
                    getWritePointer (0); // the buffer is single channel
63
64              int delayPos = delayPositionVector[channel];
65
66              delayData[delayPos] = transferVector[channel];
67
68              if (++delayPos >= delayBufferVector[channel].
                    getNumSamples())
69                  delayPos = 0;
70
71              delayPositionVector[channel] = delayPos;
72          }
73      }
74      // if more channels than network order, mix pairs of high order
            channels
75      // until order == number of channels
76      if (_channelSetting > _networkOrder)
77      {
78          int diff = _channelSetting - _networkOrder;
79          int start_index = _channelSetting - diff * 2;
80
81          for (int num = diff - 1; num < 0; --num)
82          {
83              int idx = start_index + num;
84              float *const writeOut = buffer.getWritePointer (idx);
85              float *const writeIn1 = buffer.getWritePointer (idx +
                    num);
86              float *const writeIn2 = buffer.getWritePointer (idx +
                    num + 1);
87
88              for (int i = 0; i < numSamples; ++i)
89              {
90                  writeIn1[i] = writeOut[i];
91                  writeIn2[i] = writeOut[i];
92              }
93          }
94      }
95  }
```

Die Funktion erhält als Parameter *buffer* und *delayBufferVector*, wobei es sich um den

Input/Output Buffer von JUCE handelt [2], sowie einen Vektor von Buffern für die Realisierung der Delays [3].

Die grobe Struktur der Implementierung des Delays entstammt dem Audioplugin-Tutorial von JUCE [4]. Mit zwei geschachtelten Schleifen wird über alle Samples eines Audioblocks [5] und alle Kanäle iteriert(Zeilen 26/29+60). Zuerst wird das Eingangssample zum verzögerten Abtastwert addiert (Zeile 70). Nun werden die beiden Filter auf den verzögerten Atastwert angewandt (Zeile 42). Die Filter stammen aus der DSP-Bibliothek von JUCE. Nach der Filterung wird verzögerte Abtastwert zurück in den IO-Buffer kopiert, wobei noch die Verstärkung w angewandt wird (siehe Abb. 5 bzw. Zeile 46). Zu dem Sample im IO-Buffer wird noch das Input-Sample gewichtet mit d addiert. Damit ist der Vorwärtspfad des Netzes beendet. Der verzögerte Abtastwert wird zur Vorbereitung der Multiplikation mit der Rückkopplungsmatrix abschließend in einen Vektor kopiert (Zeile 52).

Nach Durchlauf der inneren Schleife enthält dieser Vektor alle Abtastwerte des Rückkopplungspfades eines Zeitpunktes für alle Kanäle. Dieser Vektor wird nun mit der Rückkopplungsmatrix multipliziert. Hier wird die Multiplikation $\mathbf{A}\mathbf{s}(n) \circ \mathbf{g}$ ausgeführt via der schnellen Walsh-Hadamard-Transformation (siehe Abschnitt 2.4) effizient berechnet (Zeile 56).

Nach der Multiplikation werden die Daten zurück in den Delaybuffer geschrieben (Zeilen 62-66). Schließlich wird noch der Writepointer des Delaybuffers inkrementiert bzw. bei einem Überlauf zurück auf Null gesetzt (Zeilen 68/69. Diese Operationen werden wiederholt, bis alle Abtastwerte eines Blocks abgearbeitet wurden.

Vor und nach der eigentlichen Schleife des Rückkopplungsnetzwerkes befindet sich jeweils ein Schleifenkonstrukt, das Situationen abfängt, in denen der/die BenutzerIn die Netzwerkordnung niedriger als die Anzahl der I/O-Kanäle gewählt hat (siehe Abschnitt 3.2.1 bzw. Zeilen 7-24 und 76-93).

3.1.2 Parameteradaption

Da es unnötigen Vorbereitungsaufwand innerhalb der Verarbeitung produzieren würde, alle Parameter des Algorithmus für jeden Kanal bei der Berechnung jedes Audiodaten-Blockes neu zu setzen, bietet das *AudioProcessor* Objekt von JUCE eine Funktion namens *parameterChanged* an, mit der bestimmte Operationen angestoßen werden können, wenn der/die BenutzerIn einen Parameter ändert.

```
1 void FdnReverbAudioProcessor::parameterChanged (const String &
    parameterID, float newValue)
```

2. Input und Output eines VST-Plugins sind in JUCE durch ein einziges Bufferobjekt realisiert, aus dem die Daten entnommen und nach erfolgter Bearbeitung wieder zurück abgelegt werden können.

3. Das *AudioBuffer*-Objekt von JUCE unterstützt zwar beliebig viele Kanäle, jedoch lässt sich nur eine Bufferlänge pro Objekt wählen. Da die Delaylängen der einzelnen Kanäle unterschiedlich sind, ist es nicht möglich, alle Kanaldelays mit einem einzigen Objekt bereitzustellen, es ist eine Buffer-Instanz pro Kanal notwendig.

4. https://github.com/WeAreROLI/JUCE/tree/master/examples/audio%20plugin%20demo/Source

5. Die Audiodaten werden von der VST-API in Blöcken aus dem Buffer der Audioquelle bereitgestellt.

```
2 {
3     userChangedSettings = true;
4 }
```

Diese Funktion setzt eine Variable auf *true*, woraufhin vor dem Abarbeiten des nächsten Audiodaten-Blocks in der Funktion *updateParameterSettings* die neuen Parameter aus der graphischen Benutzeroberfläche in die jeweiligen Datenstrukturen übertragen werden. Die dem/der BenutzerIn zur Verfügung stehenden Parameter finden sich in Abschnitt 3.2.

```
1  inline void FdnReverbAudioProcessor::updateParameterSettings ()
2  {
3      if (userChangedSettings)
4      {
5          sampleRate = getSampleRate();
6          indices = indexGen (_networkOrder, int (*delayLength / 10.f
               ), int (*delayLength));
7
8          directGain = 1.f - *wet;
9          delayGain = *wet;
10
11         for (int channel = 0; channel < _networkOrder; ++channel)
12         {
13             // update multichannel delay parameters
14             int delayLenSamples = delayLengthConversion (channel);
15             delayBufferVector[channel].setSize(1, delayLenSamples,
                   true, true);
16             if (delayPositionVector[channel] >= delayBufferVector[
                   channel].getNumSamples())
17                 delayPositionVector[channel] = 0;
18
19             // reverberation time -> channel gain
20             feedbackGainVector[channel] = channelGainConversion (*
                   revTime, channel, delayLenSamples, sampleRate);
21
22             // update shelving filter parameters
23             highShelfFilters[channel].setCoefficients (
24                 IIRCoefficients::makeHighShelf (
25                     sampleRate, *highCutoff, *highQ,
26                     channelGainConversion (
27                         *highRevTime, channel, delayLenSamples,
                               sampleRate)));
28
29             lowShelfFilters[channel].setCoefficients (
30                 IIRCoefficients::makeLowShelf (
31                     sampleRate, *lowCutoff, *lowQ,
32                     channelGainConversion(
33                         *lowRevTime, channel, delayLenSamples,
                               sampleRate)));
34         }
35         userChangedSettings = false;
36     }
37 }
```

Weiters enthalten sind die Berechnung der Verteilung der Delaylängen [6] (Zeilen 14-17) und die Berechnung der Koeffizienten für die Filter (Zeilen 23-33).

Da diese Funktion unter Umständen die Länge der Delaybuffer verkürzt, muss sie anschließend sicherstellen, dass der Writepointer des Buffers nach der Verkürzung nicht auf einen Speicherbereich verweist, der nicht mehr Teil des Buffers ist. Tritt dieser Fall ein, so wird der Pointer auf den Beginn des Buffers gesetzt (Zeilen 16 und 17).

3.1.3 Verteilung der Delaylängen

Um unerwünschte Kammfiltereffekte zu minimieren, ist es notwendig, dass die Längen der Delays möglichst nicht auf ein Vielfaches der Länge eines der anderen Delays fallen. Eine einfach umzusetzende Möglichkeit dies zu erreichen ist, für die Delaylängen Primzahlen heranzuziehen.

Zu diesem Zweck enthält das Plugin einen einfachen Primzahlengenerator, der einen Vektor mit einem beliebig langen Segment der Primzahlenreihe beginnend mit der Zahl 3 generiert. Die nicht auf Effizienz ausgelegte Umsetzung ist hier unproblematisch, weil diese Funktion nur einmal bei der Initialisierung des Programmes aufgerufen wird. Von einer optimaleren Implementierung (z.B. Sieb des Eratosthenes) wurde daher abgesehen.

```
1  std::vector<int> FdnReverbAudioProcessor::primeNumGen (int count)
2  {
3      std::vector<int> series;
4
5      int range = 3;
6      while (series.size() < count)
7      {
8          bool is_prime = true;
9          for (int i = 2; i < range; i++)
10         {
11             if (range % i == 0)
12             {
13                 is_prime = false;
14                 break;
15             }
16         }
17
18         if (is_prime)
19             series.push_back (range);
20
21         range++;
22     }
23     return series;
24 }
```

Damit der Nachhall als diffus und nicht als Abfolge diskreter Echos wahrgenommen wird, muss zumindest ein Teil der Delaylängen eine geringe Spreitzung aufweisen.

6. Eine ausführlichere Beschreibung der Berechnung der Delaylängen befindet sich in Abschnitt 3.1.3.

Um diesem Umstand Folge zu tragen, wurden einige Implementationen getestet, wobei die unten beschriebene schließlich subjektiv am Besten abschnitt.

```
1  std::vector<int> FdnReverbAudioProcessor::indexGen (int nChannels,
       int firstIncrement, int finalIncrement)
2  {
3      std::vector<int> indices;
4
5      if (firstIncrement < 1)
6          indices.push_back (1.f);
7      else
8          indices.push_back (firstIncrement);
9
10     float increment;
11     int index;
12
13     for (int i = 1; i < nChannels; i++)
14     {
15         increment = firstIncrement + abs (finalIncrement -
               firstIncrement) / float (nChannels) * i;
16
17         if (increment < 1)
18             increment = 1.f;
19
20         index = int (round (indices[i-1] + increment));
21         indices.push_back (index);
22     }
23     return indices;
24 }
```

Die Funktion generiert einen Vektor von Indizes. Dazu muss die Länge des gewünschten Vektors, das Startinkrement und das Endinkrement spezifiziert werden. Dazwischen nimmt das Inkrement linear zu. Ist das spezifizierte Startinkrement kleiner als eins, so werden alle Inkremente unter eins auf eins gesetzt. Damit ist sichergestellt, dass nicht mehrere Kanäle die gleiche Delaylänge aufweisen.

Mit diesen Indizes werden schließlich Zahlen aus dem zuvor generierten Primzahlenvektor entnommen. Schließlich erfolgt noch die Umrechnung von Primzahl in Länge des Delays in Samples. Dies geschieht nach folgendem einfachen Zusammenhang:

$$m_i = \frac{p_i}{10} \text{ in ms} \tag{9}$$

$$m_i = \frac{m_i \text{ in ms}}{1000} \cdot f_s \tag{10}$$

```
1  inline int FdnReverbAudioProcessor::delayLengthConversion (int
       channel)
2  {
3      // we divide by 10 to get better range for room size setting
4      float delayLenMillisec = primeNumbers[indices[channel]] / 10.f;
5      return int (delayLenMillisec / 1000.f * sampleRate); //convert
           to samples
```

```
6 }
```

Der Prozess ist in Abbildung 6 graphisch für eine (fiktive) Kanalzahl von 11, ein finales Inkrement von 10 (wird über die GUI von dem/der BenutzerIn gewählt) und ein Startinkrement von 1 (wird automatisch auf $\frac{1}{10}$ des finalen Inkrements gesetzt) veranschaulicht.

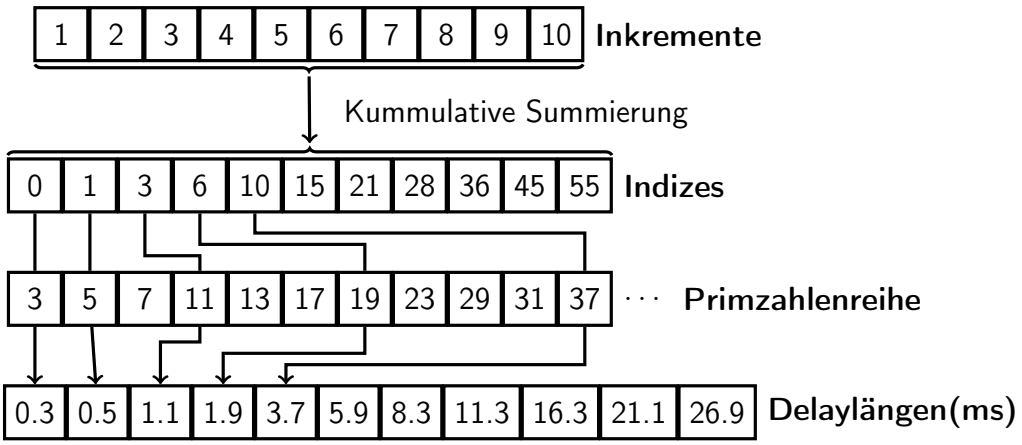

Abbildung 6 – Berechnung der Delaylängen - Aus Platzgründen ist die Primzahlenreihe nur bis zum zehnten Element abgebildet, der Prozess setzt sich aber analog der ersten 5 abgebildeten Selektionsprozesse fort.

3.1.4 Dry/Wet Regelung

Die Gewichtung zwischen verhalltem und direktem Signal weist einen linearen Zusammenhang auf, das bedeutet, dass das Signal am Ausgang über die Formel

$$\mathbf{y}(n) = w\mathbf{s}(n) + (1-w)\mathbf{x}(n) \tag{11}$$

gebildet wird, wobei $\mathbf{s}(n)$ das Signal am Ausgang des Rückkopplungsnetzwerkes darstellt. w kann von dem/der BenutzerIn über die graphische Benutzeroberfläche eingestellt werden.

3.1.5 Rückkopplungsmatrix

Um die Stabilität des Netzwerkes gewährleisten zu können, muss die Verstärkung der gesamten Schleife in allen Frequenzbereichen kleiner oder zumindest gleich eins sein. Damit dies sichergestellt ist bzw. damit man nicht separat die Verstärkung der Matrix kompensieren muss, ist es hilfreich, wenn die Matrix unitär ist (sofern die Matrix reell ist, spricht man von Orthonormalität). Vereinfacht ausgedrückt bedeutet das, dass die Matrix dem System weder Energie entzieht noch hinzufügt.

In der Praxis haben sich Hadamard-Matrizen beim Einsatz für Rückkopplungsnetzwerke bewährt, da sie einige sehr günstige Eigenschaften in sich vereinen. Hadamard-Matrizen

sind orthogonal, sie erzeugen durch den konstanten Betrag von $\frac{1}{\sqrt{N}}$ in jedem Eintrag eine maximale Diffusität im Rückkopplungsnetzwerk, und die Multiplikation lässt sich über die schnelle Walsh-Hadamard-Transformation sehr effizient implementieren (siehe Abschnitt 2.4).

Das vorliegende VST-Plugin benutzt eine Open Source Implementierung der FWHT, die frei auf GitHub zur Verfügung steht [sJH14]. Es wurde lediglich die Normierung abgeändert ($\frac{1}{\sqrt{N}}$ statt wie ursprgl. $\frac{1}{N}$).

3.1.6 Rückkopplungsverstärkung

Der Vektor der Rückkopplungsverstärkungen $\mathbf{g}$ ergibt sich aus Gleichung 7.

```
1  inline float FdnReverbAudioProcessor::channelGainConversion (float
       reverbTime,int channel, int delayLenSamples, int sampleRate)
2  {
3      // reverberation time -> channel gain
4      double gain;
5      double t = double(reverbTime);
6      gain = -60.0 / (20.0 * t);
7      gain = pow (10.0, gain);
8      double length = double(delayLenSamples) / double(sampleRate);
9      gain = pow (gain, length);
10     return float(gain);
11 }
```

Die Funktion *channelGainConversion* errechnet für eine gegebene Nachhallzeit eine geschätzte Rückkopplungsverstärkung pro Kanal in Abhängigkeit der jeweiligen Delaylänge.

3.1.7 Filterbank

Die Filterbank dient dazu, die Nachhallzeit frequenzabhängig gestalten zu können. Es kommen dabei Tiefpass- sowie Hochpass-Kuhschwanzfilter zum Einsatz. Die Verstärkung der einzelen Kanalfilter bei der Grenzfrequenz errechnet sich dabei mit der selben Funktion, die schon in Abschnitt 3.1.6 beschrieben wird. Die Details der Implementierung der Filter sind in Abschnitt 3.1.1, die Erstellung der Filterkoeffizienten in Abschnitt 3.1.2 ersichtlich.

3.2 Graphische Benutzeroberfläche

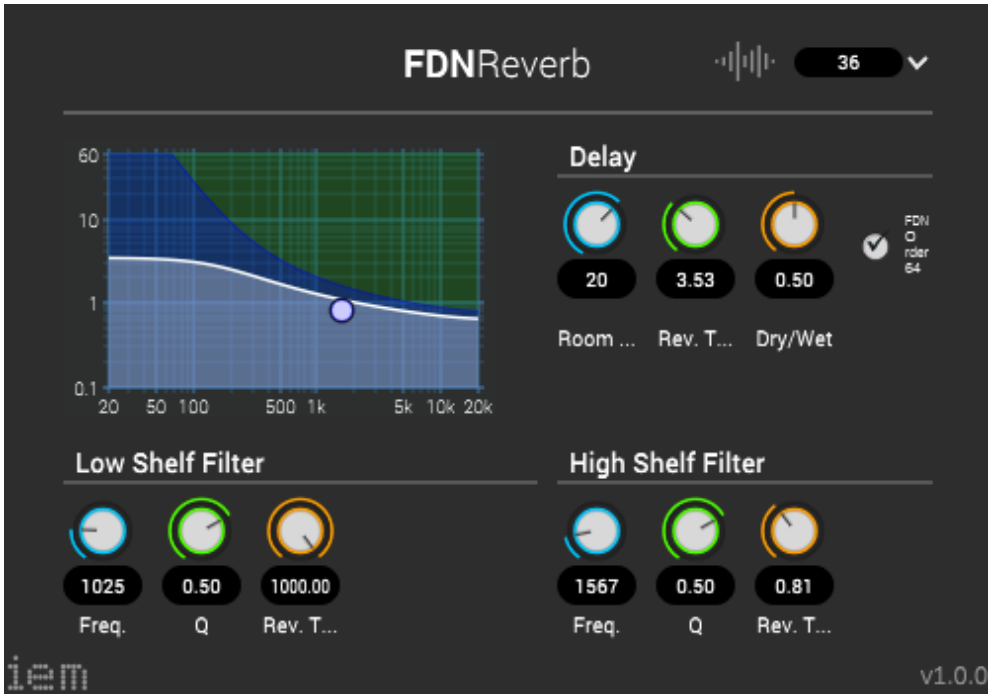

Abbildung 7 – Überblick über die graphische Benutzeroberfläche

3.2.1 Titelleiste

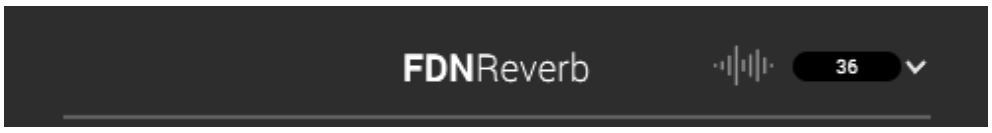

Abbildung 8 – Titelleiste

In der Titelleiste befindet sich ein Dropdown-Menü, über das die Anzahl der Ein- und Ausgänge ausgewählt werden kann. Da die Ordnung des Netzwerkes aufgrund klanglicher sowie implementationstechnischer Aspekte auf 36 oder 64 Kanäle limitiert ist, bietet diese Schaltfläche zusätzliche Flexibilität bei Verwendung anderer Ambisonics-Ordnungen.

Abbildungen 9 und 10 zeigen den Signalfluss, wenn die Anzahl der Ein- und Ausgänge K kleiner bzw. größer als die Ordnung des Netzwerkes N ist.

Bei $K > N$ werden die höchsten Kanalordnungen paarweise in jeweils einen Kanal gemischt, um so die Anzahl auf die Ordnung zu reduzieren. Am Ausgang werden diese Signale dann dupliziert, um wieder auf die korrekte Zahl an Ausgängen zu kommen.

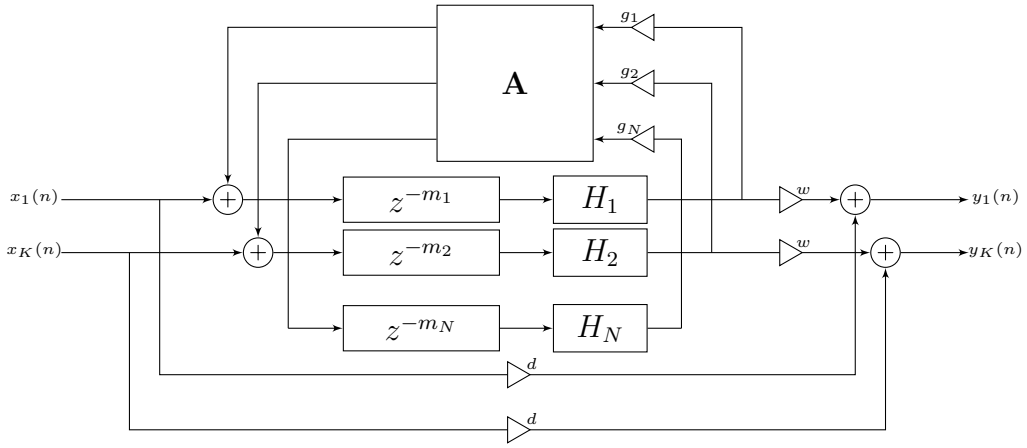

Abbildung 9 – Netzwerk mit reduzierter Ein- und Ausgangszahl

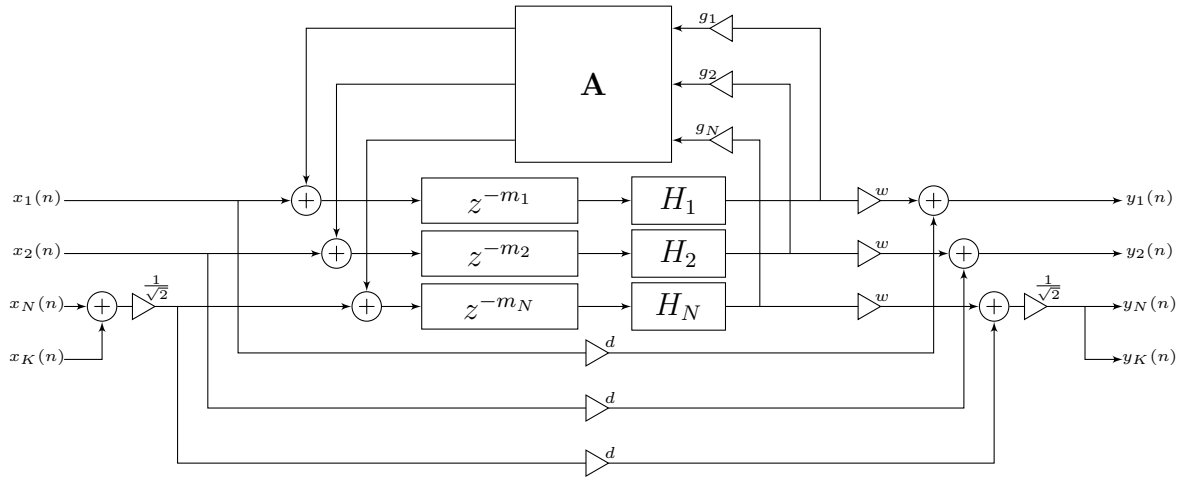

Abbildung 10 – Netzwerk mit erhöhter Ein- und Ausgangszahl ($K = N + 1$)

Das Dropdown-Menü setzt dabei eine Variable, die festlegt, wieviele Ein- und Ausgänge mit dem Netzwerk verbunden sind. Der Mechanismus beim Ändern der Kanalzahl ist analog zum Parameterupdate, sodass die Variable nicht während der Ausführung eines Blockes sondern nur davor geändert wird.

3.2.2 Delay-Sektion

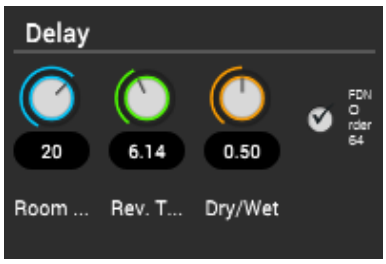

Abbildung 11 – Delay-Sektion

Die Delay-Sektion enthält drei Drehregler sowie einen Toggle-Button. Die Drehregler steuern die Länge der Delays[7], die frequenzunabhängige Nachhallzeit[8] sowie das Verhältnis zwischen verhalltem und direktem Signal[9].

Mit dem Toggle-Button lässt sich die Ordnung des Netzwerkes reduzieren, um CPU-Last einzusparen. Wird der Toggle-Button betätigt, so ruft der Audioprozess eine Funktion auf, die die Änderung der Ordnung vornimmt.

```
1  inline void FdnReverbAudioProcessor::updateChannelSetting()
2  {
3      if (userChangedChannelSettings)
4      {
5          _channelSetting = int(*channelSetting);
6          userChangedChannelSettings = false;
7      }
8
9      if (userChangedNetworkOrder)
10     {
11         if (_networkOrder < networkOrder)
12         {
13             const int diff = networkOrder - _networkOrder;
14
15             for (int i = 0; i < diff; i++)
16             {
17                 delayBufferVector.push_back (AudioBuffer<float>());
18                 delayPositionVector.push_back(0);
19
20                 highShelfFilters.push_back (IIRFilter());
```

7. Siehe Abschnitt 3.1.3
8. Siehe Abschnitt 3.1.6
9. Siehe Abschnitt 3.1.4

```
21                  lowShelfFilters.push_back (IIRFilter());
22
23                  transferVector.push_back (0.f);
24                  feedbackGainVector.push_back (0.f);
25              }
26          }
27          else
28          {
29              delayBufferVector.resize (networkOrder);
30              delayPositionVector.resize (networkOrder);
31
32              highShelfFilters.resize (networkOrder);
33              lowShelfFilters.resize (networkOrder);
34
35              transferVector.resize (networkOrder);
36              feedbackGainVector.resize (networkOrder);
37          }
38
39          _networkOrder = networkOrder;
40
41          // call updateParameterSettings
42          userChangedSettings = true;
43          userChangedNetworkOrder = false;
44      }
45  }
```

3.2.3 Hochpassfilter-Sektion

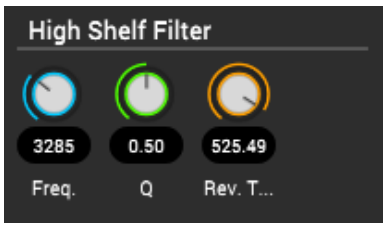

Abbildung 12 – Hochpass-Sektion

Die Hochpass-Sektion setzt sich aus drei Drehreglern zusammen. Im Detail sind das ein Regler, der die Grenzfrequenz des Filters in Hertz wählt, ein Regler für die Güte sowie ein Regler, der die abgeschätzte Nachhallzeit bei der Grenzfrequenz in Sekunden wählt.

3.2.4 Tiefpassfilter-Sektion

Analog zur Hochpass-Sektion besteht auch die Tiefpass-Sektion aus drei Drehreglern, jeweils einem für Grenzfrequenz, Güte sowie Nachhallzeit.

Abbildung 13 – Tiefpass-Sektion

3.2.5 Visualisierung der Nachhallzeit

Um die aktuell gewählten Einstellungen für den/die BenutzerIn zu veranschaulichen, enthält die Benutzeroberfläche einen interaktiven Graphen, der eine Schätzung der Nachhallzeit T_{60} über den humanperzeptiv relevanten Frequenzbereich von 20Hz bis 20kHz darstellt.

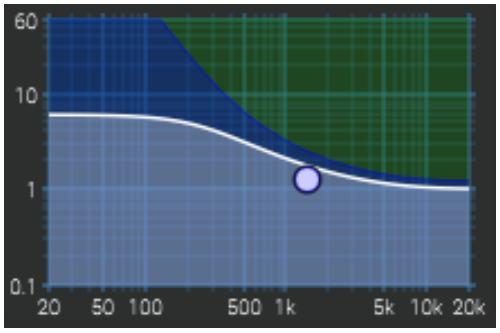

Abbildung 14 – Graph der Nachhallzeit

Der Einfluss des Tiefpassfilters (grün), des Hochpassfilters (blau) sowie die Gesamtverstärkung inklusive des Einflusses der Rückkopplungsverstärkung (weiss) sind getrennt erkennbar (siehe Abbildung 14). Alternativ zu den Drehreglern der Filter bietet der Graph die Möglichkeit, die Grenzfrequenzen der beiden Filter sowie die Verstärkung durch Bewegen zweier farbiger Punkte mit der Maus einzustellen.

Der Graph an sich ist Bestandteil der JUCE Ambisonics Library des IEM und wurde von einem Graphen zur Visualisierung von Filterkennlinien von Daniel Rudrich abgeleitet. In der GUI-Komponente selbst wird der Graph mit den relevanten GUI-Reglern verknüpft, eine eigene Funktion sorgt dann dafür, dass im Falle einer Änderung durch den/die BenutzerIn der Graph neu gezeichnet wird.

```
1 void FdnReverbAudioProcessorEditor::sliderValueChanged(Slider*
    slider)
2 {
3     if (slider == &highCutoffSlider ||
4         slider == &highQSlider ||
5         slider == &highRevTimeSlider)
6     {
```

```
7           float gain = pow (10.0, -60.0 / (20.0 * highRevTimeSlider.
               getValue()));
8
9           *highpassCoeffs = *IIR::Coefficients<float>::makeHighShelf
               (48000, highCutoffSlider.getValue(), highQSlider.
               getValue(), gain);
10
11          fv.repaint();
12      }
13      else if (slider == &lowCutoffSlider ||
14                   slider == &lowQSlider ||
15                   slider == &lowRevTimeSlider)
16      {
17          float gain = pow (10.0, -60.0 / (20.0 * lowRevTimeSlider.
               getValue()));
18
19          *lowpassCoeffs = *IIR::Coefficients<float>::makeLowShelf
               (48000, lowCutoffSlider.getValue(), lowQSlider.getValue
               (), gain);
20
21          fv.repaint();
22      }
23      else if (slider == &revTimeSlider)
24      {
25          float gain = pow (10.0, -60.0 /(20.0 * revTimeSlider.
               getValue()));
26          fv.setOverallGain (gain);
27
28          fv.repaint();
29      }
30  }
```

4 Ausblick

In der vorliegenden Arbeit wurde die Implementierung eines Rückkopplungsnetzwerk für die Verwendung in Systemen mit mehrfachen Ein- und Ausgängen vorgestellt. Das Plugin umfasst derzeit eine effiziente Implementierung des Rückkopplungsnetzwerkes, die Möglichkeit die Ordnung zu reduzieren, um CPU-Last einzusparen, eine Regelung für variable Ein- und Ausgangskanalzahlen, variable Verteilung der Delaylängen sowie eine frequenzabhängig einstellbare Nachhallzeit. Damit ist es möglich, einen flexiblen diffusen Nachhall zu erzeugen. Obwohl die erzielbaren Ergebnisse wohlklingend sind, ist der Funktionsumfang nicht ausreichend, um einen realistischen räumlichen gesamten Nachhalleindruck zu erzeugen, da die diskreten frühen Reflexionen nicht berücksichtigt werden.

Für optimale Ergebnisse sollte entweder das Plugin selbst mit frühen Reflektionen erweitert werden oder die Funktionalität durch Kombination mit anderen Plugins aus der IEM Ambisonics Bibliothek ergänzt werden.

In Sachen Effizienz besteht noch ein wenig Optimierungspotential, da z.B. aktuell immer alle Filter berechnet werden, auch wenn sie aufgrund der Wahl der Koeffizienten keinen Einfluss auf den Klang haben. Der Kuhschwanz-Hochpass ist in der Praxis wohl eher nur für spezielle Anwendungen relevant und könnte daher im Normalbetrieb deaktiviert werden.

Bei einfachen Versuchen zur Sicherstellung der Funktion des Plugins hat sich außerdem herausgestellt, dass bei ambisonischer Zuspielung unter bestimmten Quellwinkeln (z.B. $\{\Theta = 0°, \Phi = 45°\}$) unerwünschte Flattereffekte im Nachhall wahrnehmbar werden. Es wäre wünschenswert, diese störenden Laufzeiteffekte in Zukunft zu eliminieren.

Appendix

A Eine kurze Einführung in das Erstellen von VST-Plugins mit dem JUCE-Framework unter Linux

Dank einiger neuer Tools sowie der Tatsache, dass Reaper (inoffiziell) unter Linux nativ läuft und auch seit kurzer Zeit VST-Plugins im .so Format einbinden kann, stehen dem/der geneigten EntwicklerIn als Alternative zu proprietären Toolchains nun abgesehen von Reaper selbst eine Reihe von kostenlosen Open Source Applikationen zur Verfügung, die finanzielle Hürde, um in das Entwickeln von VST-Plugins einzusteigen ist damit sehr gering geworden. Die Tatsache, dass JUCE mittlerweile keine Abhängigkeit von der VST SDK aufweist, erleichtert zudem die Lizenzierung.

Mit diesem Appendix möchte ich meine (bescheidenen) Erfahrungen in der VST-Plugin-Entwicklung mit einer Open Source Toolchain festhalten und so hoffentlich zukünftigen AspirantInnen den Einstieg ein wenig erleichtern beziehungsweise das Entwickeln abseits proprietärer Softwarelösungen schmackhaft machen.

Meine Ausführungen in diesem Abschnitt beziehen sich auf eine frische Installation der Linux-Distribution Fedora in der Version 27, sie lassen sich jedoch mit geringfügigen Linux-Kentnissen und ein paar Minuten Internet-Recherche auf andere gebräuchliche Linux-Distributionen umlegen.

A.1 Setup

Bevor man mit dem Erstellen eines VST-Plugins beginnen kann, müssen zuerst die Voraussetzungen dafür geschaffen werden. So benötigt man einen C++14 kompatiblen Compiler (zumindest C++11, wenn man das DSP-Modul von JUCE nicht nutzen möchte), zudem hat JUCE Abhängigkeiten von Bibliotheken, die typischerweise nicht mit einer Standard-Distribution mitinstalliert werden.

Weiters benötigt man einen Texteditor bzw. eine Entwicklungsumgebung, um den Quelltext zu editieren und auch das Plugin zu debuggen. Es gibt eine vielzahl von Möglichkeiten, angefangen bei reinen Texteditoren wie vim in Verbindung mit dem GNU Debugger bis zu der von JUCE unterstützten IDE Code::Blocks. Die Auswahl obliegt primär dem Geschmack beziehungsweise den Erfahrungen des/der EntwicklerIn, ich persönlich verwende aktuell den Open Source Texteditor/IDE Microsoft Visual Code inklusive des dazugehörigen C++-Plugins in Kombination mit der Konsole zum Starten des Compilers, da ich mit der Bedienung gut vertraut bin und die in Visual Code integrierte graphische Benutzeroberfläche für den GNU Debugger leicht zu bedienen ist.

Meine Ausführungen beziehen sich auf das Kompilieren mittels *make*, es gibt weiters noch die möglichkeit unter Linux Code::Blocks zu verwenden und von JUCE ein dazugehöriges Projekt anlegen zu lassen. Wenn man mit Code::Blocks vertraut ist, lässt sich diese Anleitung vermutlich auch mit einem Code::Blocks Projekt kombinieren.

A.1.1 Compiler und Abhängigkeiten

Der folgende Konsolenbefehl installiert gcc-c++ inklusiver aller Abhängigkeiten von JUCE für den Projucer sowie das Erstellen von VST-Plugins:

```
$ sudo dnf install gcc-c++ freetype-devel libcurl-devel libX11-devel
    libXext-devel libXinerama-devel webkitgtk4-devel gtk3-devel alsa-
    lib-devel libGL-devel
```

Damit ist der Compiler bereit, um mit JUCE erstellte Programme zu kompilieren. Die hier angegebenen Namen der Bibliotheken beziehen sich auf RPM Repositories, für apt basierte Distributionen (z.B. Ubuntu) lauten die Namen ein wenig anders. Im Zweifelsfall ist es meistens schon Zielführend, einfach den Namen der Linux-Distribution sowie den Namen der Bibliothek bei Google einzugeben, damit man den Namen des jeweiligen Paketes herausfindet.

Dies ist eine Liste der Namen der Bibliotheken: [10]

- freetype2
- libcurl
- x11
- xext
- xinerama
- webkit2gtk-4.0
- gtk+-x11-3.0
- alsa
- libgl-dev

Es stehen unter Linux auch andere Compiler wie clang zur Verfügung, ich beschränke mich in dieser Arbeit jedoch auf g++.

A.1.2 JUCE

Unter Linux empfiehlt es sich, direkt mit Git [11] das Entwickler-Repository auszuchecken, da es sich anschließend sehr einfach gestaltet, JUCE auf dem neuesten Stand zu halten.

Dazu navigiert man die Konsole an den Ordner, in dem das JUCE-Repository liegen soll. In meinem Fall wäre dies z.B. */home/username/Documents/repos/*.

Dort führt man dann den Befehl

```
$ git clone https://github.com/WeAreROLI/JUCE.git
```

10. Unter anderen Distributionen ergibt sich möglicherweise eine andere Liste von Abhängigkeiten, diese Liste kann also per se nicht den Anspruch erheben, vollständig zu sein. Sie stellt außerdem den Stand zum Zeitpunkt des Verfassens dar. Fehlende Abhängigkeiten werden vom Compiler als Fehlermeldung geliefert, sollte also eine Abhängigkeit fehlen, kann man eine Internet-Suche mit der Ausgabe des Compilers durchführen.

11. Falls der werte Leser mit Git noch nicht vertraut ist, so sei ihm dieses Tool wärmstens ans Herz gelegt. Eine eingehendere Einführung in Git würde den Rahmen dieser Arbeit maßlos sprengen, es empfiehlt sich jedoch unbedingt, zumindest eine Form von Versionskontrolle für das Entwickeln unter C++ zu verwenden. Als Einstieg eignet sich beispielsweise https://git-scm.com/book/en/v2.

aus. Das Repository sollte nun von Git in den Ordner JUCE heruntergeladen werden.

Wenn man JUCE updaten will, kann man einfach in den JUCE ordner navigieren und dort die neuesten änderungen *pullen*.

```
$ cd /home/username/Documents/repos/JUCE
$ git pull
```

Sollte dies notwendig werden, kann man auch sehr einfach zwischen dem Entwicklungsbranch sowie dem Releasebranch von JUCE hin- und herwechseln, mit

```
$ cd /home/username/Documents/repos/JUCE
$ git checkout develop
```

bringt man das Repository auf den neuesten Entwicklungsstand und mit

```
$ git checkout master
```

wieder zurück auf die neueste Release-Version.

A.1.3 Projucer kompilieren

Um ein JUCE-Projekt zu erstellen, empfiehlt es sich, das von JUCE mitgelieferte Utility *Projucer* zu verwenden. Der Projucer erstellt Code-Templates für die jeweilige Art von Projekt und ist vor allem insofern hilfreich, als dass er für das Projekt plattformübergreifend Compiler-Direktiven erstellen kann. Damit wird es spielend leicht, ein Projekt, das mit Linux erstellt wurde auf einem Windows PC für Windows oder auf einem Mac für OSX zu kompilieren .

Zuerst muss unter Linux der Projucer jedoch selbst erst kompiliert werden [12]. Dazu navigiert man in den Projektfolder des Projucers und ruft das dort liegende Makefile folgendermaßen auf:

```
$ cd ../JUCE/extras/Projucer/Builds/LinuxMakefile
$ make CONFIG=Release AR=gcc-ar -j 6
```

CONFIG=Release sagt dem Compiler, dass er für die Release-Konfiguration kompilieren soll, das bedeutet, dass keine Debug-Symbole in den Code kompiliert werden und dass der Compiler einen hohen Optimierungsgrad verwendet, dadurch ist anschließend die Performance des resultierenden Programmes besser.

AR=gcc-ar bezieht sich auf den zu verwendenden Archiver. JUCE hat mit der aktuellen Version [13] Link-Time-Optimizations (LTOs) unter Linux standardmäßig in der Releasekonfiguration aktiviert, gcc kompiliert mit dieser Einstellung jedoch nur, wenn als Archiver gcc-ar angegeben wird.

Mit *-j* wird dem Compiler mitgeteilt, wieviele Threads er für das Kompilieren und Linken verwenden soll. Als Faustregel empfiehlt es sich, die Anzahl der zur Verfügung stehen-

12. Diesen Schritt sollte man nach jedem Update von JUCE wiederholen, damit es nicht zu Inkompatibilitäten kommt. Sollte eine Inkompatibilität vorliegen, weist der Projucer jedenfalls darauf hin.
13. Aktuelle Version von JUCE zum Zeitpunkt des Verfassens: 5.2

den Prozessorkerne mit 1.5 zu multiplizieren. Für kleinere Programme verschwindet der Vorteil aufgrund des Overheads allerdings.

Wenn der Compiler fertig ist, befindet sich das Programm unter *../JUCE/extras/Projucer/Builds/LinuxMakefile/build/Projucer*.

A.2 Erstellen eines Audioplugin-Projektes

Nachdem der Projucer kompiliert wurde, ist alles bereit um ein JUCE Projekt anzulegen.

Dazu öffnet man den Projucer mit

```
$ cd ../JUCE/extras/Projucer/Builds/LinuxMakefile
$ ./Projucer
```

Beim erstmaligen Öffnen erscheint ein Login-Bildschirm, dort kann man einen Entwickleraccount anlegen. Für Kleinstunternehmer mit weniger als 50000$ Jahresumsatz beziehungsweise für Studenten/Lehrende ist der Zugang kostenlos.

Ist man angemeldet, öffnet sich der Projucer mit der Seite, um neue Projekte anzulegen (Abb. 15).

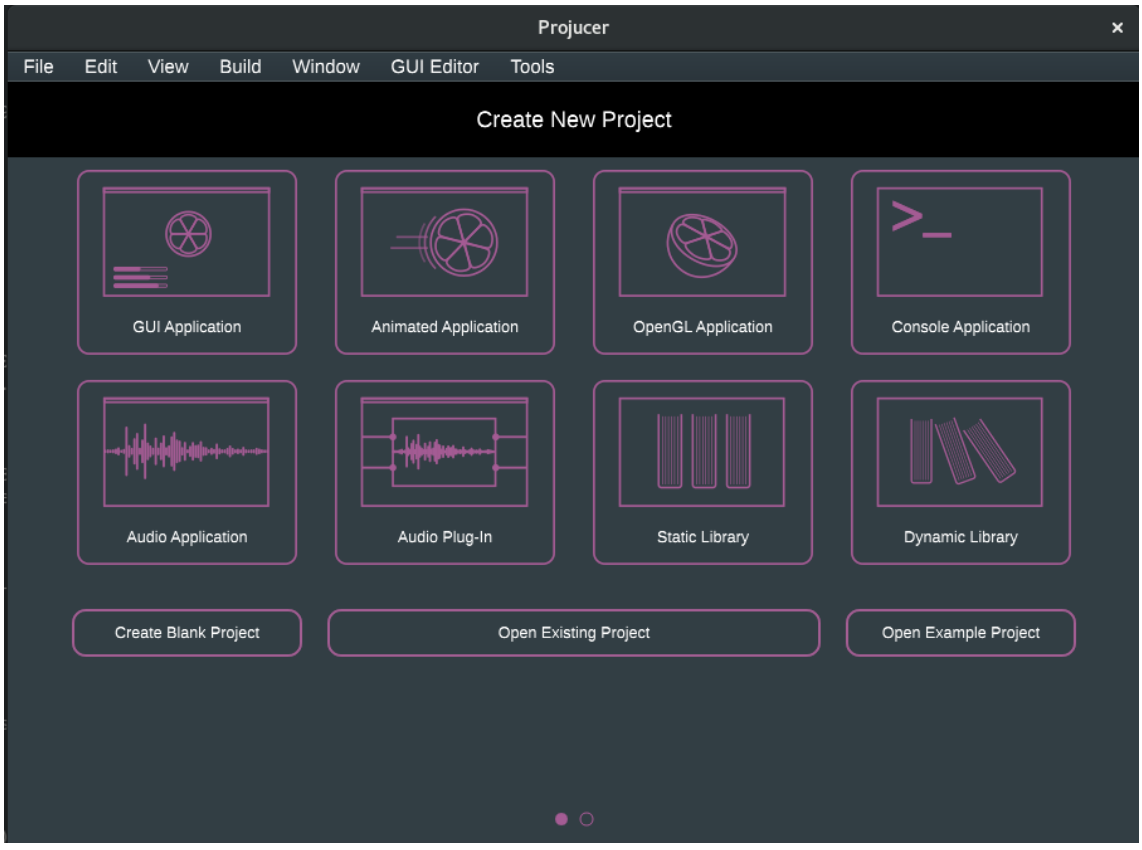

Abbildung 15 – Startseite zum Anlegen neuer Projekte

Klickt man auf Audio-Plugin, wird man zur zweiten Seite der Plugin-Erstellung weitergeleitet (Abbildung 16).

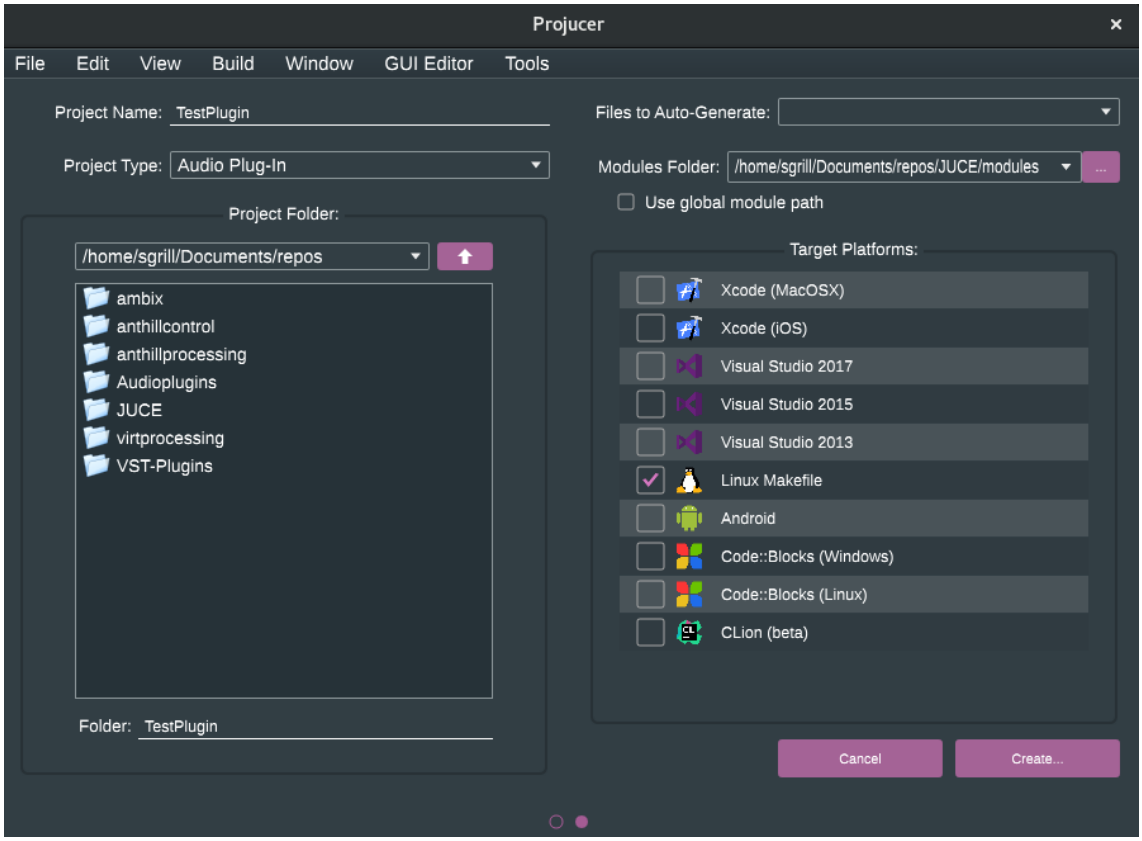

Abbildung 16 – Startseite 2 zum Abschließen der Plugin-Erstellung

Hier kann man festlegen, in welchem Ordner das Projekt erstellt werden soll. Rechts oben muss man den Pfad zur JUCE Bibliothek angeben (Modules Folder). Der Pfad muss auf den Ordner *modules* im JUCE Hauptordner verweisen. Darunter kann man festlegen, für welche Zielplattformen Compiler-Direktiven erstellt werden sollen. Die Auswahl lässt sich jederzeit erweitern, falls man sich entschließen sollte, eine zusätzliche Plattform unterstützen zu wollen. In jedem Fall muss für die Linux-Plattform ein Häkchen bei „Linux Makefile“ gesetzt werden.

Mit „Create...“ wird die Erstellung des Plugins abgeschlossen und man gelangt zur Hauptseite des Projekts (Abbildung 17).

Über das Zahnradsymbol im oberen Teil dieses Fensters gelangt man zu den Einstellungen, hier kann man wählen welche Plugin-Formate der Compiler erstellen soll (VST, AU, RTAS, Standalone, etc.). Außerdem kann man hier den Namen der Firma (z.B. IEM) eintragen, die Versionsinformation des Plugins editieren, den C++ Standard ändern, etc. Die Positionen sind selbsterklärend, man sollte allerdings die Liste kurz durchgehen und gemäß den eigenen Präferenzen ausfüllen. In jedem Fall muss für das Erstellen von VST-Plugins „Build VST“ angehakt sein.

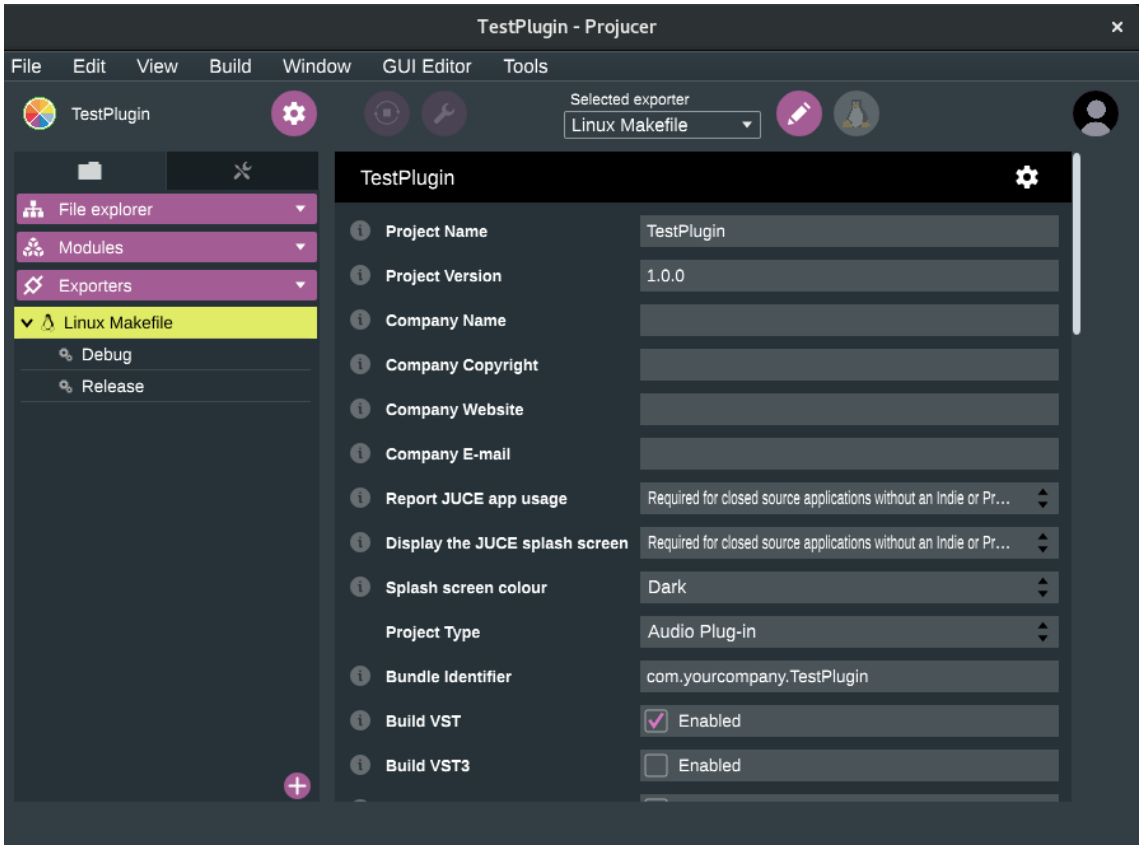

Abbildung 17 – Hauptseite des Projektes

Die Seite enthält noch einige andere wichtige Funktionen. Unter dem Reiter „Exporters“ lassen sich weitere Plattformen hinzufügen (z.B. Windows oder OSX). Außerdem lassen sich bestehende Compiler-Direktiven editieren (etwa der Optimierungsgrad unter der „Release“-Einstellung, etc.). Für den Anfang ist ausreichend, die Einstellungen zu belassen.

Der Reiter „Modules“ steuert, welche Module von JUCE vom Compiler eingebunden werden. Um die Filegröße des resultierenden Plugins zu reduzieren, empfiehlt es sich, nur Module einzubinden, die tatsächlich benötigt werden. Für die ersten Schritte sind die standardmäßig aktivierten Module völlig ausreichend.

Über den obersten Reiter im linken Menü („File explorer“) kann man Dateien mit Quellcode zu dem Projekt hinzufügen. Diese Option ist insbesondere dann wichtig, wenn man bestimmte Funktionen oder Objekte in separate Dateien auslagern will. Header-Dateien können auch direkt über relative oder absolute Includes miteinbezogen werden, .cpp Dateien muss man allerdings im „File explorer“ eintragen. Die vom Projucer erstellten Files sind bereits standardmäßig hinzugefügt, somit ist alles bereit, um mit dem Programmieren der Funktionalität zu beginnen.

Mit *File->Save Projekt* werden die aktuellen Einstellungen für das Projekt übernommen, danach kann man den Projucer schließen.

A.3 Aufbau eines JUCE-Audioplugins anhand eines Lautstärkereglers

Dieses Kapitel stellt die wichtigsten Codebausteine eines Audioplugins in JUCE anhand eines kurzen Beispiels vor. Der volle Funktionsumfang hat im Rahmen dieser Arbeit nicht annähernd Platz, daher sollte dieser Abschnitt nur als kurze Einleitung betrachtet werden.

Um einen umfassenderen Überblick zu erhalten, empfiehlt sich ein Studium der Tutorials auf der Webseite von JUCE[14], die Online-Dokumentation aller Objekte[15] sowie unbedingt die Referenzimplementierungen von diversen Applikationen. Letztere befinden sich unter *../JUCE/examples/*. Für den Anfang ist die Audio Plugin Demo sicher am geeignetsten. „AudioAppExample“ und „DSPDemo“ enthalten fortgeschrittene Konzepte zu den Themen Benutzeroberfläche sowie dem DSP-Modul von JUCE. Die Implementierung des in dieser Arbeit vorgestellten Plugins basiert zu weiten Teilen auf Konzepten aus diesen Quellen.

A.3.1 Kompilieren des Projektes

Um die Korrektheit aller Projekteinstellungen zu überprüfen, sollte man das Projekt einmal kompilieren.

Dazu navigiert man die Konsole mit

14. https://juce.com/tutorials
15. https://juce.com/doc/classes

```
$ cd ../TestPlugin/Builds/LinuxMakefile
```

zum Makefile. Das Projekt kann man dann mit dem Befehl

```
$ make CONFIG=Release AR=gcc-ar -j 6
```

kompilieren. Eine Erklärung der *make*-Parameter befindet sich in Abschnitt A.1.3. Will man das Projekt in der Debug-Konfiguration kompilieren, wird obiger Befehl einfach zu

```
$ make -j 6
```

Damit sollte das Projekt zum ersten mal kompilieren. Wählt man die Release-Konfiguration, werden einige Warnungen aus der zlib angezeigt, diese brauchen uns jedoch nicht weiter zu stören.

Man kann das Projekt auch schon versuchsweise in Reaper einbinden, dazu fügt man dem Plugin-Suchpfad von Reaper einfach den Projektordner hinzu. Reaper wählt automatisch immer die jüngste Version des Plugins (falls man zwischen Release und Debug hin- und herwechseln möchte).

Das Plugin schleust zu diesem Zeitpunkt nur Audiodaten durch, um das zu ändern, widmet sich der nächste Abschnitt der AudioProcessor Klasse.

A.3.2 AudioProcessor Klasse

Die AudioProcessor Klasse ist der Teil des Plugins, in dem die Signalverarbeitung stattfindet, und damit unsere erste Anlaufstelle für unseren Lautstärkeregler.

Dankenswerterweise erstellt der Projucer beim Anlegen des Projektes bereits einiges an Code, den wir nurmehr geringfügig modifizieren müssen, damit daraus ein Lautstärkeregler wird.

Zu allererst benötigen wir im AudioProcessor Objekt eine Variable, die unseren Gain-Wert speichert. Dazu legen wir im *private*-Block von *PluginProcessor.h* eine Variable an (Zeile 5). Der *private*-Block des Headerfiles sollte nun in etwa so aussehen:

```
1 private:
2     //==============================================================================
3     JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (TestPluginAudioProcessor)
4
5     float *gain;
6 };
```

Da wir jetzt eine Variable mit unserem Gain haben, wollen wir es mit den Audiodaten multiplizieren. Uns interessiert daher die Funktion *processBlock* in der Datei *PluginProcessor.cpp*, die jedesmal aufgerufen wird, wenn ein neuer Audiodatenblock verarbeitet werden soll. Für das Verständnis der Verarbeitung in JUCE ist es entscheidend zu wissen, dass die Funktion *processBlock* keinen Buffer zurückgibt, sondern stattdessen die Daten aus dem Buffer modifiziert und sie dann wieder in dem selben Buffer ablegt.

Praktischerweise bietet uns das Template, das der Projucer angelegt hat, bereits alles, was wir benötigen, um die Daten zu modifizieren. Die Schleife in Zeile 18 iteriert über alle Kanäle und in Zeile 20 bekommen wir einen Pointer auf die Daten des jeweiligen Kanals.

```
1  void TestPluginAudioProcessor::processBlock (AudioSampleBuffer&
       buffer, MidiBuffer& midiMessages)
2  {
3      ScopedNoDenormals noDenormals;
4      const int totalNumInputChannels  = getTotalNumInputChannels();
5      const int totalNumOutputChannels = getTotalNumOutputChannels();
6
7      // In case we have more outputs than inputs, this code clears
           any output
8      // channels that didn't contain input data, (because these aren
           't
9      // guaranteed to be empty - they may contain garbage).
10     // This is here to avoid people getting screaming feedback
11     // when they first compile a plugin, but obviously you don't
           need to keep
12     // this code if your algorithm always overwrites all the output
           channels.
13     for (int i = totalNumInputChannels; i < totalNumOutputChannels;
           ++i)
14         buffer.clear (i, 0, buffer.getNumSamples());
15
16     // This is the place where you'd normally do the guts of your
           plugin's
17     // audio processing...
18     for (int channel = 0; channel < totalNumInputChannels; ++
           channel)
19     {
20         float* channelData = buffer.getWritePointer (channel);
21
22         // ..do something to the data...
23     }
24 }
```

Wir modifizieren die Schleife folgendermaßen:

```
1  // This is the place where you'd normally do the guts of your
       plugin's
2  // audio processing...
3  int numSamples = buffer.getNumSamples();
4  for (int channel = 0; channel < totalNumInputChannels; ++channel)
5  {
6      float* channelData = buffer.getWritePointer (channel);
7
8      // ..do something to the data...
9      for (int idx = 0; idx < numSamples; ++idx)
10     {
11         channelData[idx] *= *gain;
12     }
13 }
```

Eine eingebettete Schleife (Zeile 9-12) iteriert jetzt über alle Samples und multipliziert diese mit unserem Gain [16] (Zeile 11).

Zum jetzigen Zeitpunkt haben wir allerdings noch keine Möglichkeit, das Gain zu verändern. Dazu müssen wir ein Interface erstellen, über das die Audio-Software, die das Plugin einbindet mit dem Plugin selbst kommunizieren kann. Unsere Benutzeroberfläche wird dieses Interface später ebenfalls nutzen.

JUCE bietet einige relativ leicht zu benutzende Möglichkeiten, solch ein Interface aufzubauen. Ich werde hier den *ValueTreeState* beschreiben. Je nach Anwendung sind unter Umständen andere Lösungen eleganter, es ist daher für moderat fortgeschrittene EntwicklerInnen empfehlenswert, sich einen Überblick zu verschaffen.

Um den *ValueTreeState* verwenden zu können, muss man die Klassendeklaration in der Headerdatei folgendermaßen erweitern.

```
1 class TestPluginAudioProcessor  : public AudioProcessor, public
      AudioProcessorValueTreeState::Listener
```

Listener sind in JUCE typischerweise Callbacks, mit denen man im Falle der Änderung eines Parameters eine bestimmte Funktion aufrufen kann. Beim *ValueTreeState* löst der Listener eine Synchronisation einer Variable des *AudioProcessor* mit einer Variable der GUI aus wenn der/die UserIn diese ändert. GUI und Audioverarbeitung laufen in separaten Threads. Will man also eine Kommunikation zwischen beiden Komponenten herstellen, so muss diese *threadsafe* sein, das heißt, die GUI darf nicht zu einem beliebigen Zeitpunkt eine Variable des Audiothreads ändern, da dies möglicherweise mitten in der Abarbeitung eines Audioblockes passieren könnte und zu Problemen wie Artefakten oder gar einem Absturz führen kann. Das FDN-Plugin enthält Beispiele von Strategien, wie man dieses Problem lösen kann [17], für unsere Implementierung ist eine Änderung des Gains aber zu keinem Zeitpunkt problematisch.

Im *public*-Block des Headers müssen wir jetzt einen *ValueTreeState* und eine zusätzliche Funktion deklarieren. Wir fügen also folgende Zeilen im *public*-Block hinzu.

```
1 AudioProcessorValueTreeState parameters;
2 void parameterChanged (const String &parameterID, float newValue)
      override;
```

Zuletzt müssen wir den Konstruktor unseres *AudioProcessor* in *PluginProcessor.cpp* modifizieren und die neu deklarierte Funktion *parameterChanged* definieren (sie hat für uns keine Funktion, muss aber definiert werden).

```
1 TestPluginAudioProcessor::TestPluginAudioProcessor()
2 #ifndef JucePlugin_PreferredChannelConfigurations
```

16. Diese Implementierung ist weder besonders performant noch elegant, sie dient nur der Veranschaulichung. Das Buffer-Objekt hat die Methode applyGain, mit der ein Gain auf alle Werte in dem Buffer angewendet werden kann (siehe https://juce.com/doc/classAudioBuffer). In der Praxis würde man natürlich diese Methode verwenden. Zusätzlich müsste man Sorge tragen, dass sich zwischen einzelnen Samples das Gain nicht zu stark ändern kann, da es sonst zu hörbaren Artefakten kommen würde (dazu gibt es bestehende Funktionen in JUCE).

17. Siehe zum Beispiel die Funktion *updateParameterSettings* sowie die Steuerung des Programmflusses in *processBlock*.

```
3           : AudioProcessor (BusesProperties()
4                             #if ! JucePlugin_IsMidiEffect
5                             #if ! JucePlugin_IsSynth
6                             .withInput  ("Input",  AudioChannelSet::
                                 stereo(), true)
7                             #endif
8                             .withOutput ("Output", AudioChannelSet::
                                 stereo(), true)
9                             #endif
10                            )
11  #endif
12  parameters (*this, nullptr)
13  {
14      parameters.createAndAddParameter ("gain", "Gain", "",
15                                        NormalisableRange<float> (0.0
                                             f, 1.0f, 0.01f), 1.0f,
16                                        [](float value) {return
                                             String (value);}, nullptr)
                                             ;
17
18      parameters.state = ValueTree (Identifier ("TestPlugin"));
19
20      parameters.addParameterListener ("gain", this);
21
22      gain = parameters.getRawParameterValue ("gain");
23  }
```

Die Funktionsdefinition kommt unter die bestehenden Funktionsdefinitionen in der .cpp Datei.

```
1  void TestPluginAudioProcessor::parameterChanged (const String &
       parameterID, float newValue)
2  {
3  }
```

Damit haben wir dem öffentlichen Interface unseres Plugins einen Parameter „gain“ hinzugefügt und ihn mit der privaten Variable *gain* verknüpft. Wir können das Plugin mit

```
$ make CONFIG=Release AR=gcc-ar -j 6
```

kompilieren und anschließend in Reaper einbinden. Es hat noch keine eigene GUI, jedoch sieht Reaper die öffentlichen Parameter des Plugins und erstellt eine interne Bedienoberfläche (Abbildung 18).

Das Plugin ist zu diesem Zeitpunkt funktionstüchtig und gewichtet Audiodaten mit dem eingestellten Gain. Allerdings haben wir nicht die Möglichkeit die GUI, die Reaper für das Plugin erstellt in irgendeiner Form zu gestalten, daher widmen wir uns im nächsten Abschnitt wie wir die Benutzeroberfläche unseres Plugins mit dem Regler verknüpfen.

Davor müssen wir jedoch noch dem *ProcessorEditor* Zugriff auf den *ValueTreeState* ermöglichen. Dazu modifizieren wir die *createEditor* Funktion unseres *AudioProcessor* indem wir ihm zusätzlich die Variable *parameters* übergeben:

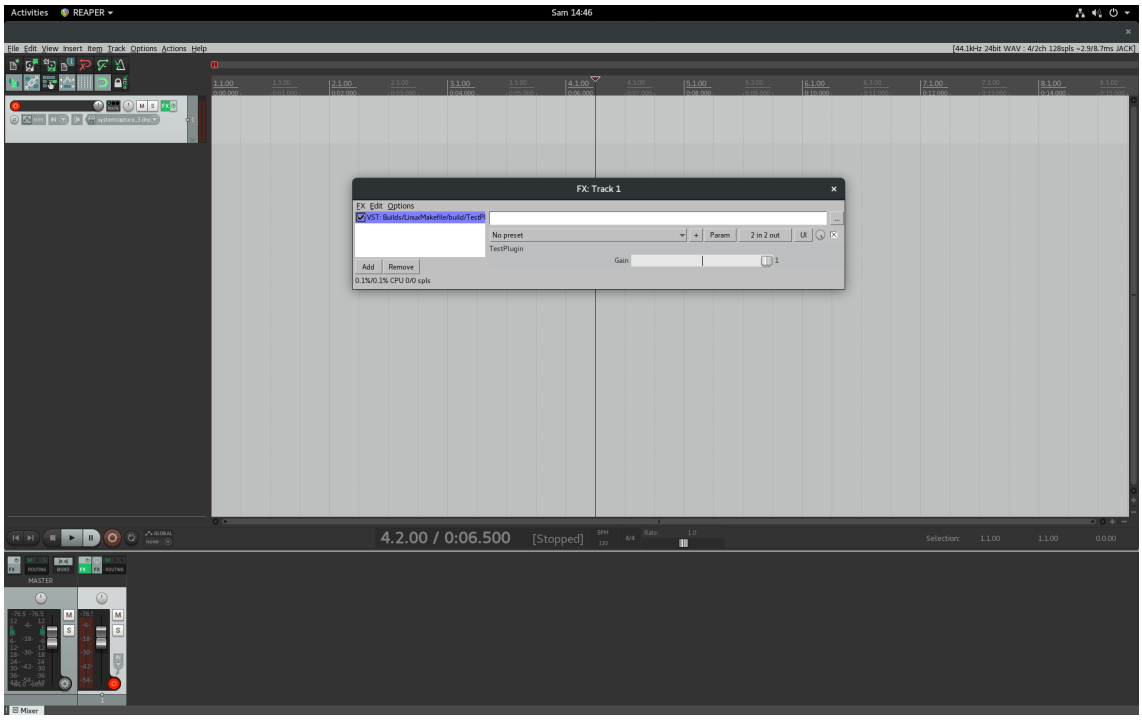

Abbildung 18 – Das Plugin mit Benutzeroberfläche von Reaper

```
1  AudioProcessorEditor* TestPluginAudioProcessor::createEditor()
2  {
3      return new TestPluginAudioProcessorEditor (*this, parameters);
4  }
```

A.3.3 AudioProcessorEditor Klasse

Um die GUI mit dem *ValueTreeState* zu verknüpfen muss *PluginEditor.h* folgendermaßen angepasst werden:

```
1  /*
2  ==============================================================================
3
4      This file was auto-generated!
5
6      It contains the basic framework code for a JUCE plugin editor.
7
8  ==============================================================================
9  */
10
11 #pragma once
12
13 #include "../JuceLibraryCode/JuceHeader.h"
14 #include "PluginProcessor.h"
```

```
15
16 typedef AudioProcessorValueTreeState::SliderAttachment
       SliderAttachment;
17 //
       ==============================================================================

18 /**
19 */
20 class TestPluginAudioProcessorEditor  : public AudioProcessorEditor
       , private Slider::Listener
21 {
22 public:
23     TestPluginAudioProcessorEditor (TestPluginAudioProcessor&,
          AudioProcessorValueTreeState&);
24     ~TestPluginAudioProcessorEditor();
25
26     //
          ==============================================================================

27     void paint (Graphics&) override;
28     void resized() override;
29
30     void sliderValueChanged(Slider* slider) override;
31
32 private:
33     // This reference is provided as a quick way for your editor to
34     // access the processor object that created it.
35     TestPluginAudioProcessor& processor;
36     AudioProcessorValueTreeState& valueTreeState;
37
38     Slider gainSlider;
39
40     ScopedPointer<SliderAttachment> gainAttachment;
41
42     JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (
          TestPluginAudioProcessorEditor)
43 };
```

Der *Editor* kann so den *ValueTreeState* übernehmen und ihn auch ändern. In zeile 38 deklarieren wir außerdem unser *Slider*-Objekt und in Zeile 40 ein *SliderAttachment*, das den Wert unseres *Slider*sin den *ValueTreeState* übernimmt.

Schließlich müssen wir noch *PluginEditor.cpp* editieren:

```
1 /*
2 ==============================================================================

3
4   This file was auto-generated!
5
6   It contains the basic framework code for a JUCE plugin editor.
7
8 ==============================================================================

9 */
```

```
10
11 #include "PluginProcessor.h"
12 #include "PluginEditor.h"
13
14
15 //==============================================================================
16 TestPluginAudioProcessorEditor::TestPluginAudioProcessorEditor (
       TestPluginAudioProcessor& p, AudioProcessorValueTreeState& vts)
17   : AudioProcessorEditor (&p), processor (p), valueTreeState (vts)
18 {
19   // Make sure that before the constructor has finished, you've set
         the
20   // editor's size to whatever you need it to be.
21   setSize (200, 200);
22
23   addAndMakeVisible (&gainSlider);
24   gainAttachment = new SliderAttachment (valueTreeState, "gain",
       gainSlider);
25   gainSlider.setSliderStyle (Slider::Rotary);
26   gainSlider.setTextBoxStyle (Slider::TextBoxBelow, false, 60, 20);
27   gainSlider.setTooltip("Gain");
28 }
29
30 TestPluginAudioProcessorEditor::~TestPluginAudioProcessorEditor()
31 {
32 }
33
34 //==============================================================================
35 void TestPluginAudioProcessorEditor::paint (Graphics& g)
36 {
37   // (Our component is opaque, so we must completely fill the
       background with a solid colour)
38   g.fillAll (getLookAndFeel().findColour (ResizableWindow::
       backgroundColourId));
39
40
41 }
42
43 void TestPluginAudioProcessorEditor::resized()
44 {
45   // This is generally where you'll want to lay out the positions
       of any
46   // subcomponents in your editor..
47
48   gainSlider.setBounds (90, 90, 90, 90);
49 }
50
51 void TestPluginAudioProcessorEditor::sliderValueChanged (Slider*
       slider)
52 {
```

```
53
54 }
```

In Zeile 23 erstellen wir unser *Slider*-Objekt und verknüpfen es in Zeile 24 mit dem *ValueTreeState*. Zeilen 25 bis 27 setzen einige der Formatparameter des Objektes.

Abschließend wird in Zeile 48 das *Slider*-Objekt auf der GUI positioniert. Unsere einfache GUI ist damit fertig und wir können das Projekt nocheinmal mit

```
$ make CONFIG=Release AR=gcc-ar -j 6
```

kompilieren. Öffnet man Reaper erneut und bindet das Plugin ein, sollte nun eine native Plugin-GUI bereitstehen, mit der man das Gain manipulieren kann (siehe Abbildung 19).

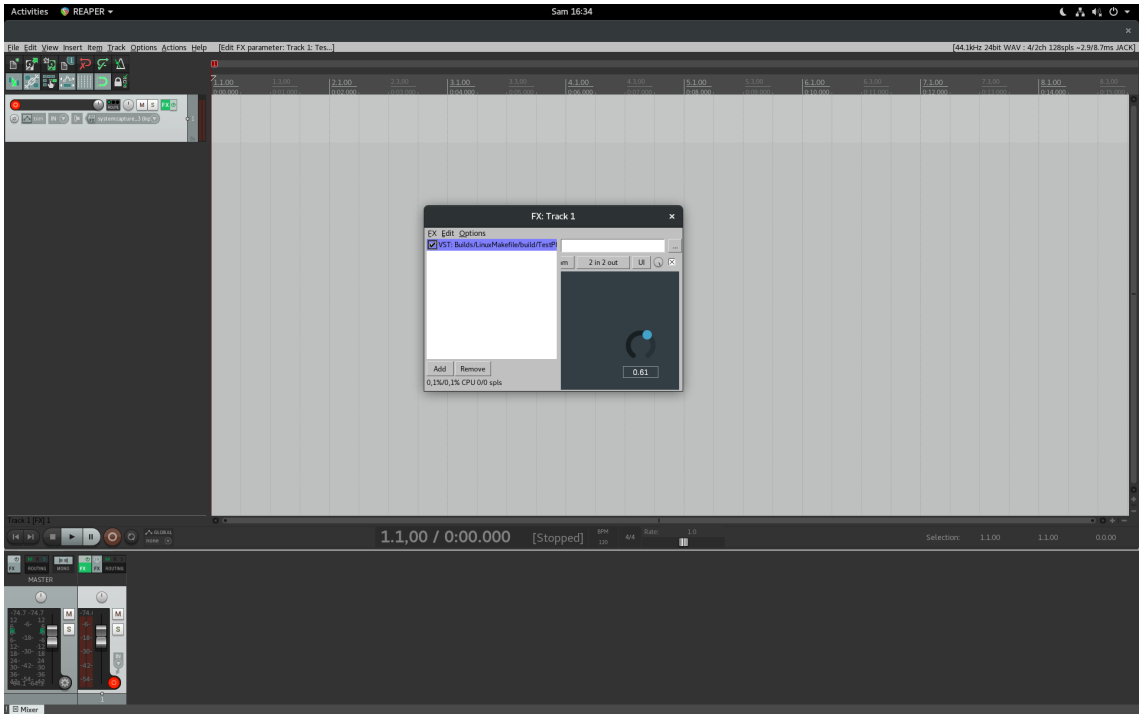

Abbildung 19 – Das Plugin mit der einfachen selbstgestalteten Benutzeroberfläche

Literatur

[Blo17] M. Blochberger, "FDN reverberation in ambisonics," 2017.

[SH15] S. J. Schlecht and E. A. P. Habets, "Time-varying feedback matrices in feedback delay networks and their application in artificial reverberation," *Journal of the Acoustical Society of America*, vol. 138, no. 3, 2015.

[SH17] ——, "Accurate reverberation time control in feedback delay networks," in *Proceedings of the 20 th International Conference on Digital Audio Effects (DAFx-17)*, 2017.

[sJH14] sheljohn (Jonathan H.), "Walsh-Hadamard," 2014. [Online]. Available: https://github.com/sheljohn/WalshHadamard

[SP82] J. Stautner and M. Puckette, "Designing multi-channel reverberators," *Computer Music Journal*, vol. 6, no. 1, 1982.