WaveFieldSynthesizer Plugin

Algorithms in Acoustics and Computer Music 02 - Seminar

Michael Reiter, Lukas Ignaz Maier
Supervisors: Dr. Franz Zotter, Dr. Matthias Frank

Graz, March 11, 2021

Institute of Electronic Music and Acoustics

Abstract

Target of this seminar project was to implement a Wave Field Synthesis VST plugin. The algorithm used in the plugin is based on and aligned with previous work.
The fundamental mathematical formula for the algorithm is the loudspeaker driving function, which consists of a filter, a delay and a gain block. The equations for these blocks are no longer valid when the virtual source position crosses the loudspeaker line. This is addressed by inverting the delay and adjusting the equations for the gain factor. Additionally the typical Doppler effect is avoided by limiting the update speed of the virtual source position for the delay line. Finally, a circular loudspeaker arrangement was implemented and derived from the calculations of the linear loudspeaker array.
The structure of the plugin is explained in this work alongside an overview over the classes and the data flow that was implemented. We did not have access to a loudspeaker array, so the plugin was tested in a binaural setup with recorded impulse responses.

Contents

1 Introduction

Wave-field synthesis aims to synthesize the wave field of a virtual primary source for a given listener position by using secondary sources, usually loudspeakers. In the present approach a 2.5D approach is used - the mathematical foundation is built on 3D theory, while the implementation uses only two dimensions.
This work begins with the basics of wave field synthesis. There the mathematical concept and the mathematical building blocks for an audio signal chain are explained. Next the audio signal chain and the two possible loudspeaker arrangements are explained in more detail.
The main goal of this work was a real-time audio plugin, which is shown in the next section. The program overview, the data flow, the GUI (graphical user interface) and the real-time aspects are depicted in more detail. The work closes with a binaural showcase and the conclusion.

2 Basics of Wave Field Synthesis

This section briefly describes the basics of the state-of-the-art wave field synthesis approach. This includes the mathematical concept and a simplification of the driving function, which leads to the building blocks for the audio signal chain.
The approach was already described in detail in a seminar paper by Lukas Gölles and Lukas Maier [3]. Additionally, this paper took great inspiration from Gergely Firtha's PhD thesis [1].

2.1 Mathematical concept

We assume a point source **x**, which serves as our (primary) virtual source. We also define a source-free volume Ω, which is limited by infinitesimally spaced secondary sound sources. These can be described by the free-field Green's function

$$G(r) = \frac{e^{-ikr}}{4\pi r}. \tag{1}$$

Here, k denotes the wave number $\frac{2\pi f}{c}$, f the frequency, c the speed of sound and r the distance between the source and the receiver. Next we assume a listener position **y** inside the volume Ω. For this listener position the Kirchhoff-Helmholtz integral is defined as

$$p(\boldsymbol{y}) = -\oint_{\partial\boldsymbol{\Omega}} \left[G(r_{\boldsymbol{s},\boldsymbol{x}})\boldsymbol{\nabla}G(r_{\boldsymbol{s},\boldsymbol{y}}) - G(r_{\boldsymbol{s},\boldsymbol{y}})\boldsymbol{\nabla}G(r_{\boldsymbol{s},\boldsymbol{x}})\right]^T d\boldsymbol{\Omega}(\boldsymbol{s}), \tag{2}$$

where $r_{\boldsymbol{s},\boldsymbol{y}} = ||\boldsymbol{s}-\boldsymbol{y}||$ is the distance between a secondary source and the listener position, and $r_{\boldsymbol{s},\boldsymbol{x}} = ||\boldsymbol{s}-\boldsymbol{x}||$ is the distance between the same secondary source and the primary source. Secondary refers to the representing sources on the contour $\partial\Omega$, while the primary source is the virtual source whose field should be reconstructed by the sources on $\partial\Omega$,

within the volume Ω. This is shown in figure 1.
∇ denotes the gradient, while $\partial\boldsymbol{\Omega}$ is a surface normal vector at the boundary $\partial\Omega$ of the volume, whose length is proportional to the size of the differential surface path $\partial\boldsymbol{\Omega} = \boldsymbol{n}\partial\Omega$.

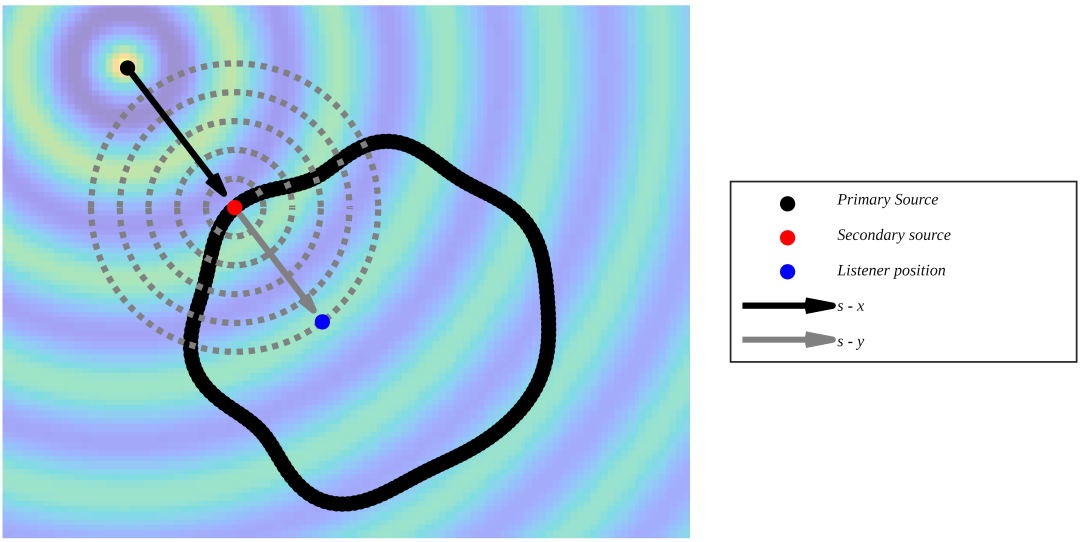

Figure 1: Concept of the wave-field synthesis, using primary source, a secondary source and the listener position [3].

We calculate the first derivative and find out that the gradient of the Green function can be approximated with

$$\boldsymbol{\nabla} r = \frac{\boldsymbol{s} - \boldsymbol{x}}{r_{\boldsymbol{s},\boldsymbol{x}}}. \tag{3}$$

This leads to a high-frequency approximation of the Kirchhoff-Helmholtz integral defined as

$$p(\boldsymbol{y}) = ik \oint_{\partial\boldsymbol{\Omega}} G(r_{\boldsymbol{s},\boldsymbol{x}}) G(r_{\boldsymbol{s},\boldsymbol{y}}) \left(\frac{\boldsymbol{s} - \boldsymbol{y}}{r_{\boldsymbol{s},\boldsymbol{y}}} - \frac{\boldsymbol{s} - \boldsymbol{x}}{r_{\boldsymbol{s},\boldsymbol{x}}} \right)^T d\boldsymbol{\Omega}(\boldsymbol{s}). \tag{4}$$

In the next step we use the stationary phase approximation [**?**] to evaluate the integral. We may do this because we may assume that we have sinusoids with rapidly oscillating phase arguments. After using the stationary phase approximation and simplifying the terms we derive the driving function

$$D = \frac{ik}{2\pi} \frac{\max\left\{(\boldsymbol{s} - \boldsymbol{x})^T \boldsymbol{n}, 0\right\}}{r_{\boldsymbol{s},\boldsymbol{x}}^2} e^{-ik\, r_{\boldsymbol{s},\boldsymbol{x}}}. \tag{5}$$

It already consists of a the filter term $\frac{ik}{2\pi}$, the gain term $\frac{\max\left\{(\boldsymbol{s}-\boldsymbol{x})^T \boldsymbol{n}, 0\right\}}{r_{\boldsymbol{s},\boldsymbol{x}}^2}$ and the delay term $e^{-ik\, r_{\boldsymbol{s},\boldsymbol{x}}}$. With the maximum operator we avoid destructive interference, which would be caused by secondary sources producing waves that propagate into the wrong direction within Ω.

2.2 Simplifying the driving function

It is possible to further simplify the driving function by using a 2.5D[1] approach. For that we neglect the interval over the y-axis. This simplifies the driving function to

$$D = \sqrt{\frac{ik}{2\pi}} \sqrt{\frac{r_{\boldsymbol{s},\boldsymbol{y}} r_{\boldsymbol{s},\boldsymbol{x}}}{r_{\boldsymbol{x},\boldsymbol{y}}}} \frac{1}{r_{\boldsymbol{s},\boldsymbol{x}}^2} \max\left\{(\boldsymbol{s}-\boldsymbol{x})^T \boldsymbol{n}(\boldsymbol{s}), 0\right\} e^{-ik\, r_{\boldsymbol{s},\boldsymbol{x}}}. \tag{6}$$

Now we can even further simplify this by assuming two kinds of reference: line and circle. For line reference, the wave-field is ideal for a reference line, which runs through the listener position and is parallel to the x-axis (see figure 7). The driving function then takes the form

$$D(\boldsymbol{s}) = \sqrt{\frac{ik}{2\pi}} \sqrt{\frac{|y_y|}{|y_y| + |x_y|}} \frac{1}{r_{\boldsymbol{s},\boldsymbol{x}}^{1.5}} (-x_y)\, e^{-ik\, r_{\boldsymbol{s},\boldsymbol{x}}}. \tag{7}$$

Note how the gain part of this driving function is independent of the x-component of the listener position.
For circle reference, the wave-field synthesis error is minimal in a half-circle around the listener position. The driving function for this form is

$$D(\boldsymbol{s}) = \sqrt{\frac{ik}{2\pi}} \sqrt{\frac{R - r_{\boldsymbol{s},\boldsymbol{x}}}{R}} \frac{1}{r_{\boldsymbol{s},\boldsymbol{x}}^{1.5}} (-x_y)\, e^{-ik\, r_{\boldsymbol{s},\boldsymbol{x}}}. \tag{8}$$

In total this leaves us with the building blocks for the 2.5D wave-field synthesis in table .

	Filter	Gain	Delay
2.5D line reference	$\sqrt{ik}$	$-x_y \sqrt{\frac{\lvert y_y\rvert}{2\pi(\lvert y_y\rvert+\lvert x_y\rvert)}} \frac{1}{r_{\boldsymbol{s},\boldsymbol{x}}^{1.5}}$	$\Delta t = \frac{r_{\boldsymbol{s},\boldsymbol{x}}}{c}$
2.5D circle reference	$\sqrt{ik}$	$-x_y \sqrt{\frac{R-r_{\boldsymbol{s},\boldsymbol{x}}}{2\pi \cdot R}} \frac{1}{r_{\boldsymbol{s},\boldsymbol{x}}^{1.5}}$	$\Delta t = \frac{r_{\boldsymbol{s},\boldsymbol{x}}}{c}$

Table 1: 2.5D wave-field synthesis blocks.

3 Signal chain

For simplicity we assume a mono input as source signal. In any other case, the current version of the implementation mixes down the channels to mono before processing. The driving signal for each loudspeaker consists of a (high-pass) filter, a gain factor and a delay. Since the filter is the same for each loudspeaker it gets applied first. Afterwards each channel gets its individual gain factor and delay depending on the position of the speaker in relation to the source and the speaker.

In [3] two different variants for calculating these parameters are proposed: The first version defines a line in parallel to the linear loudspeaker array, for which the sound field

[1] The mathematical foundation still uses a 3D approach, but we only integrate over two dimensions.

gets optimally synthesized. The second variant uses a reference circle (with every point of the circle being equidistant from the source). In this algorithm we mostly focus on the first variant and the circle reference was not included in the current implementation.

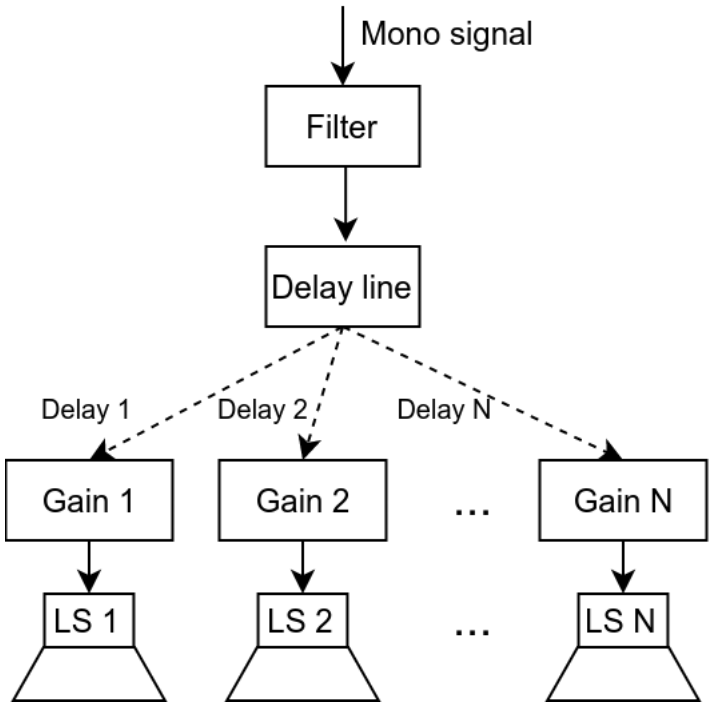

Figure 2: Signal flow from mono source to loudspeaker signal

3.1 Filter

Each loudspeaker is assigned to a point on the reference contour. The ideal synthesis line is therefore discretized. This results in a frequency limit up to which the calculations in chapter 2.2 are valid. The exact aliasing frequnecy is calculated in [2] as,

$$f_{al} = \frac{c}{\Delta s(|\sin(\max(\theta_{n,x}))| + |\sin(\max(\theta_{n,y}))|)}, \tag{9}$$

where $\theta_{n,x}$ refers to the angle between the normal vector of the synthesis line and the source and $\theta_{n,y}$ refers to the angle between the normal vector of the synthesis line and the respective loudspeaker. This results in a worst case scenario, where the aliasing frequency equals to

$$f_{al} = \frac{c}{2\Delta s} \tag{10}$$

In the test environment in [3] a loudspeaker setup with a distance of 15.1 cm between speakers was used to record impulse responses. These were recorded with an artificial dummy head, so that the loudspeaker signals could be converted to a binaural signal. This way the algorithm/plugin could be tested roughly without having the loudspeaker array

available. For this dummy-head-virtualized configuration, that is described in more detail in [3], the resulting aliasing frequency lies at around $1136Hz$. The filter in Fig. 3 was used as WFS equalization. It features a high-pass filter that decreases with approximately 3dB per octave below 1126 Hz, and it is applied to each speaker signal. For the implementation a FIR-Filter with around 70 coefficients was used.

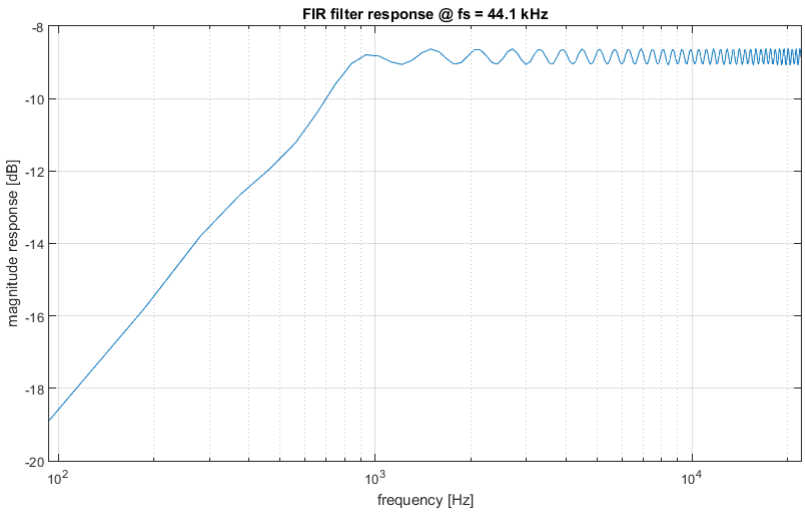

Figure 3: Magnitude spectrum of WFS equalization filter

The use of a fixed FIR-filter results in several problems which, due to time constrains, were not addressed as of the writing of this documentation:

- Different sampling frequencies
  Depending on the sampling frequency of the DAW in use, the cut-off frequency gets shifted.

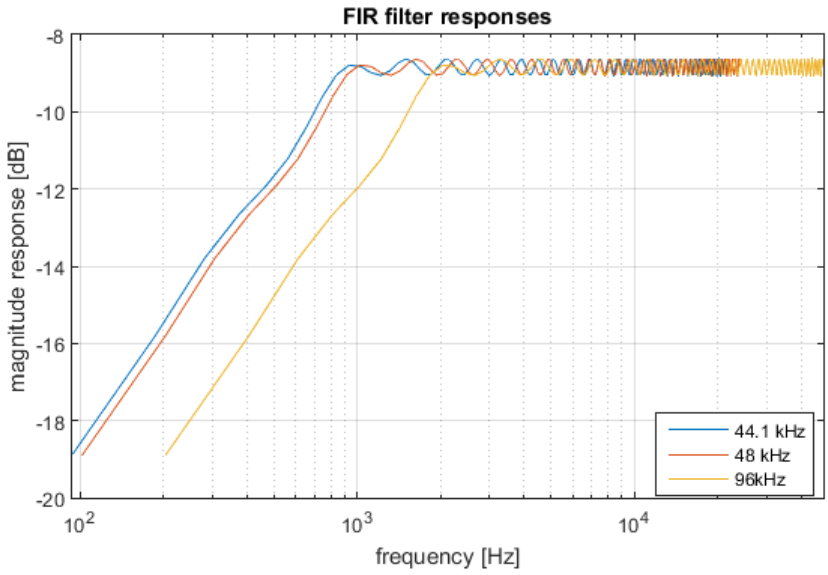

Figure 4: Magnitude spectrum of applied FIR-filter with varying sampling frequencies

- Different distance between speakers
  The cut-off frequency is only valid for a 15.1cm distance between speakers and can increase drastically if the speakers are further apart. Calculations for significantly less dense loudspeaker arrangements are not valid.

An alternative filter using an IIR architecture is proposed in [4], but was not further pursued for simplicity.

3.2 Gain

From the definition of the driving function we get an equation for the channel specific gain,

$$gain = -x_y \sqrt{\frac{|y_y|}{2\pi(|y_y| + |x_y|)}} \frac{1}{r_{s,x}^{1.5}} \tag{11}$$

where $|x_y|$ is the distance along the y-axis from the source to the speaker, $|x_y|$ is the distance along the y-axis from the listener to the speaker and $r_{s,x}$ the distance from the virtual source to the speaker.

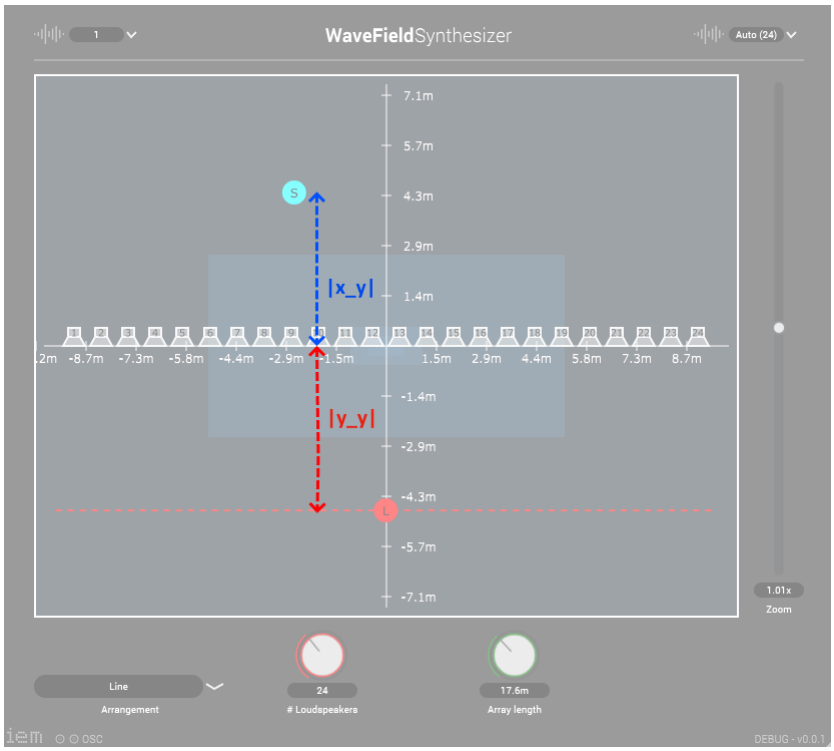

Figure 5: GUI snapshot with relevant geometric parameters drawn onto it

Since the plug-in only gets blocks of audio data and the source position does not get updated every audio sample, to provide smooth transitions within the signal blocks, a sample wise interpolation was implemented with a linear ramp. Additionally the gain has to be limited because in close proximity to the source, the speaker signal power could diverge otherwise.

Focused source

So far we needed to assume the virtual source to be 'behind' the loudspeaker array (in the positive y-half-plane) because only for this case, the calculations are perfectly valid within Ω. In case the end user of the plug-in wants to position the virtual source between listener and speaker array, as a focused source the gain factor needs to be adapted to:

$$gain = -x_y \sqrt{\frac{|y_y|}{2\pi(|y_y| - |x_y|)}} \frac{1}{r_{s,x}^{1.5}} \tag{12}$$

This way, the amplitude increases if the array line is crossed and the result sounds reasonable. Finally, if the virtual source moves even further from the loudspeaker array and goes beyond the listener, there can no longer be any reasonable synthesis at the position of the listener. For simplicity the gain is automatically muted as soon as the source enters this region.

3.3 Delay

The delay

$$\Delta t = \frac{r_{s,x}}{c} \tag{13}$$

depends on the source and the loudspeaker positions. Justas with the linear interpolation of the gain within the signal block, also the delay can be linearly interpolated to account for a change in the position of the virtual source. Sample-wise interpolation of the delay provides smooth transitions.
However, rapid movement of the virtual source position causes a Doppler effect. This effect changes the pitch of the audio source significantly, unfortunately differently so for every loudspeaker. Several ways can be formulated to counteract this issue (for a refined mathematical model see [1]). Our implementation chose a sample-wise limit for the change of delay. If the limit is chosen properly, the Doppler Effect can be mostly avoided. However, the virtual source might be lagging behind the GUI positioning of the user.

```
1  float delayInSeconds = distanceSourceSpeaker / speedOfSound;
2  float delayInSamples = delayInSeconds * sampleRate;
3  float deltaDelayInSamples = (delayInSamples - previousDelay) / (
       nSamplesInBuffer - sample);
4  deltaDelayInSamples = jlimit(-0.1f, 0.1f, deltaDelayInSamples);
5  // make it so that there can never be a change in delay by more than
       x samples from one sample to the next
6
7  delayInSamples = previousDelay + deltaDelayInSamples;
8
9  writePointerOutputChannel[sample] = delayLine.popSample(0,
       delayInSamples, updateRingBufferReadPointer);
10 previousDelay = delayInSamples;
```

Listing 1: Calculation of delay in code

In these lines of code, the (fractional) delay gets updated sample by sample. The changes in delay, compared to the previous sample, get limited (line 4) and the corresponding sample is read (line 9).

Focused source

When the virtual source moves over the line of loudspeakers toward the listener, it becomes a focused source. In order to achieve a focused point source, we need the loudspeaker signals to arrive at this position at the same time. In contrast to the calculations before, the delay to the closest loudspeaker is not the shortest but the longest one this time. The delay for a source beyond the loudspeaker array

$$\Delta t = -\frac{r_{s,x}}{c} \tag{14}$$

is now a negative value. Since the plug-in runs in real-time, negative delays need to be made causal by adding a common delay. We fixed this problem by checking all loudspeaker channels for their delay and subtracting the lowest delay from each channel. This way, the lowest delay is forced to zero. A problem with this approach is, that if the plug-in is used on multiple tracks, relative delays between tracks get uncontrolled. Thus each instance of the plug-in finds its own lowest delay but has no knowledge of the other instances. Another approach proposed was to always use an additional fixed offset-delay that compensates the 'negative' delays. This alternative would fix some issues but in this case, there would always be an additional bothersome delay. If the offset-delay is not chosen large enough, there could again be problems with a negative delay for some speakers.

4 Circular loudspeaker arrangement

As this implementation uses the formalism reviewed in [3], a circular loudspeaker array was implemented according to the formalism outlined there for filter, gain and delay. The aforementioned literature does not mention a circular arrangement of loudspeakers. So for the driving signals of each speaker the approach was adapted to fit this geometry.

First, the listener position is always assumed to be in the center of the circle. A new axis between listener and source is spanned. For the gain calculations for the circular array, the positions of listener and source along the y-axis were necessary. Here we use the same calculations but along this new axis called y_{new}:

$$gain = -x_{y_{new}} \sqrt{\frac{|y_{y_{new}}|}{2\pi(|y_{y_{new}}| + |x_{y_{new}}|)}} \frac{1}{r_{s,x}^{1.5}} \tag{15}$$

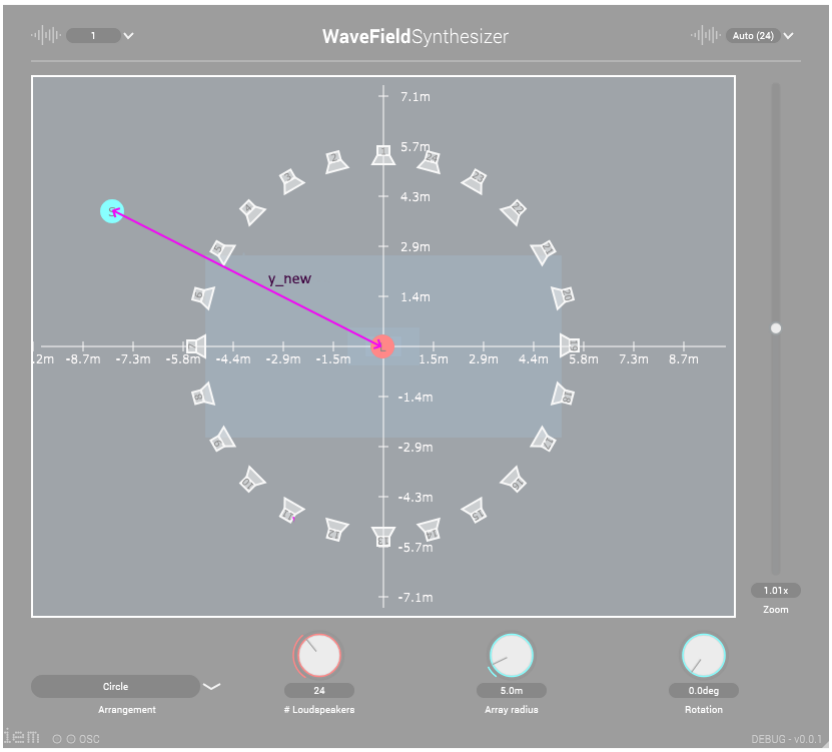

Figure 6: Circular loudspeaker arrangement

The idea is to use the same calculations as before while treating the connecting line shown in Fig. 6 like the y-axis before. Depending on the source position, the ideal synthesis line changes because it is always normal to this line. A sweet spot in the middle should persist in any case.
Another problem is that for circular arrangement speakers that are farther away from the source than the listener is, need to be disabled. If the source comes from in front of the

listener, the gain of the speakers behind the listener needs to be set to zero.
This implementation should only be considered as a draft. There are algorithms that provide accurate driving functions for the circular array. The plug-in offers the graphical user interface and set-up but the algorithmic aspect can and should be improved upon.

5 Plugin implementation

Here the main goal of this work, the *WaveFieldSynthesizer* real-time VST plugin, is explained in more detail.

5.1 Graphical User Interface

For an easy understanding of the plugin it is good to first show the GUI.

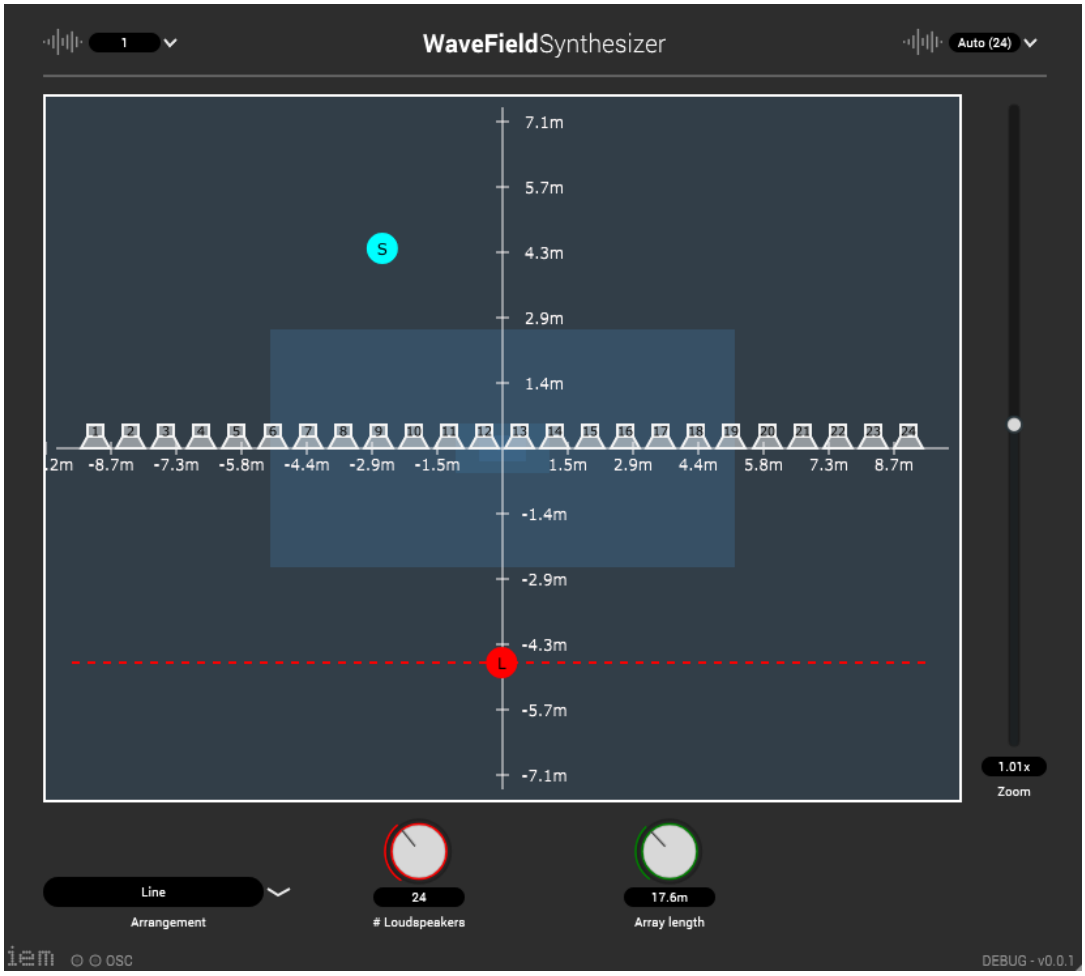

Figure 7: Screenshot of the GUI using linear loudspeaker arrangement.

The interface is divided into 3 parts: the title bar, the main area with the XYPad and the controls, and the footer with the OSC interface and the version number. Table 2 further explains the different controls for the user.

Control Element	**Action**
XYPad	Source and listener position can be dragged here. The background is static and updated only when slider values change.
Zoom	Zoom factor in the XYPad, logarithmically spaced between 0.1 and 10.
Arrangement	Dropdown menu with either line or circular arrangement.
# Loudspeakers	The number of loudspeakers in the user's arrangement. Internally the plugin limits it to the maximum number of available channels.
Array length	Total length of the loudspeaker array, measured between center of leftmost and center of rightmost loudspeakers.

Table 2: GUI controls for line arrangement

In addition to these controls there are the bus setup controls in the header bar as well as the OSC control element in the footer bar.
The array length control only applies for a loudspeaker array in line arrangement. To show the difference between the two arrangements the next figure shows the plugin using a circle arrangement.

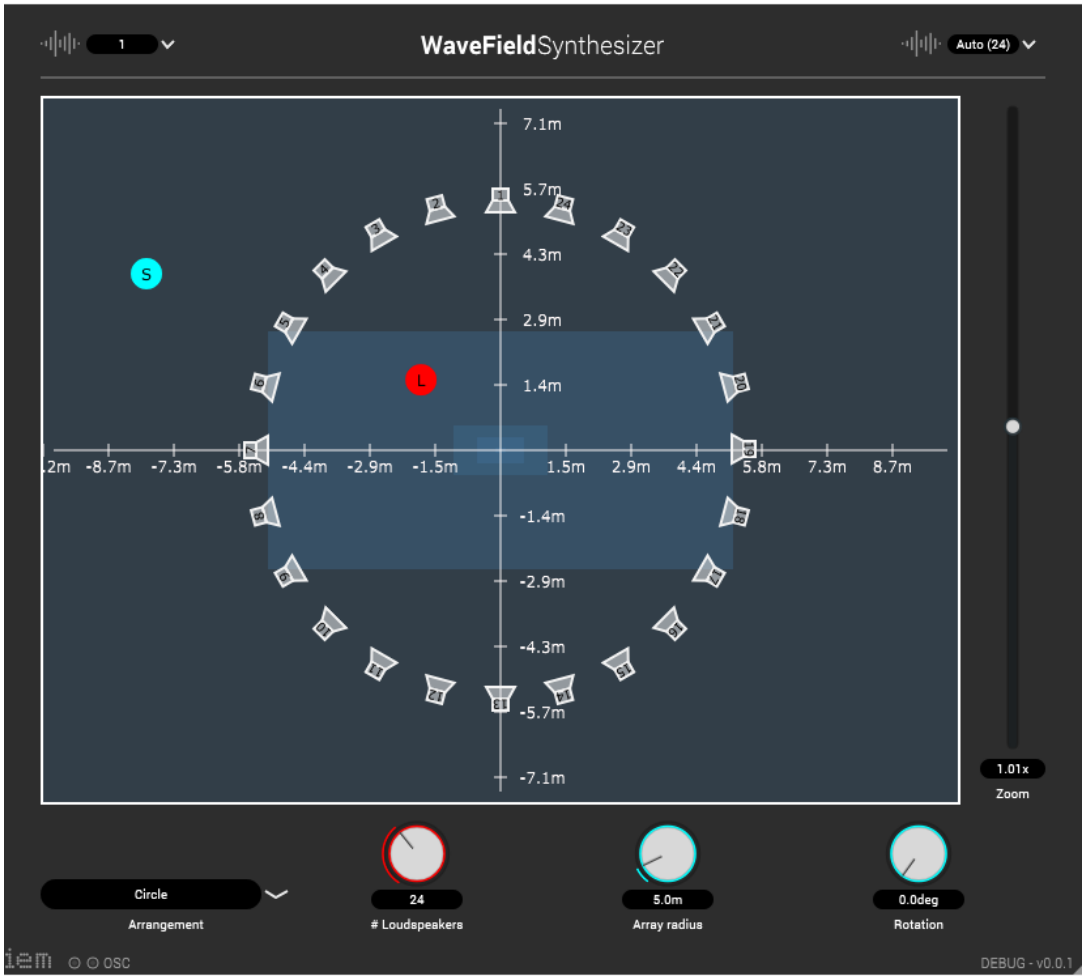

Figure 8: Screenshot of the GUI using circle arrangement.

Instead of the array length slider there is now an array radius slider which controls the loudspeaker array radius. Additionally there is a rotation control slider that can be used to give the first and otherwise frontal loudspeaker a rotational offset.

5.2 Class description

Some of the classes in this plugin are directly derived from the JUCE framework. The classes that apply specifically for this wave-field synthesis are manually designed and implemented.

WfsAudioProcessor This is the main processing class of the plugin. It is the equivalent to what is usually called the **PluginProcessor** class in JUCE. It takes care of setting up all audio-related plugin components, and of continuously receiving and processing the audio sample blocks. It also holds the audio parameter object, which is directly linked to the corresponding GUI controls. Specifically for the WFS it

also holds 3 additional objects.
The first one is the FIR-filter at the beginning of the WFS signal chain. Next is a circular buffer, where audio samples are stored. It implements a delay line by reading from it with a variable sample offset. The third is the **WfsSpeakerProcessor** array. It is used to manage the loudspeaker coordinates and to compute the gain and delay of each loudspeaker.

WfsSpeakerProcessor Each loudspeaker is represented by an object of this class. It stores the respective loudspeaker coordinates and channel, as well as the write pointer to the corresponding output channel in the main audio buffer.
It implements functions to compute delay and gain, and also to write the processed WFS signals to the output audio buffer. These functions need to be called by the **WfsAudioProcessor**.

WfsAudioProcessorEditor This class creates and updates the GUI. In the JUCE framework this class is usually called the **PluginEditor**. In addition to setting up the GUI it also manages the callbacks that get triggered each time a GUI control changes through user interaction.

WfsXyPad This class plots the source and the listener position in the GUI. It also redraws both positions if another GUI control parameter affecting the positions is changed. Additionally it highlights an active position by drawing a white circle around the position a user is hovering over.

WfsXyPadBackground This class renders the background of the XYPad in the GUI. This includes the loudspeakers, the coordinates and the blue background rectangles. Drawing these components requires a decent effort by the CPU. To avoid having this effort each time the source or the listener position are updated, the background is logically decoupled from the positions. It is only redrawn if one of the knob controls changes.

5.3 Program overview and data flow

Figure 9 shows the program overview, with all the classes, the DAW, and the data flow between the objects in the class.

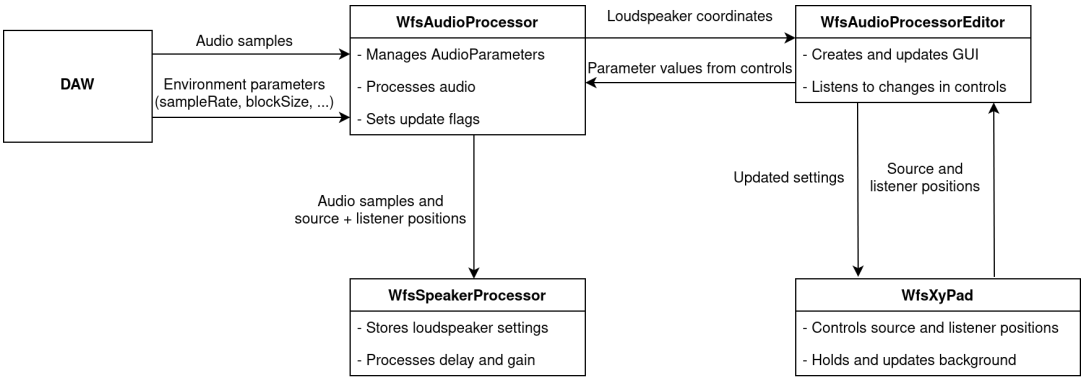

Figure 9: Program overview with all classes and data flow.

5.4 Real-time safety

The program implements some resource-consuming update tasks, e.g. updating the loudspeaker array or the GUI background. In order to ensure real-time processing these updates are handled asynchronously.

Whenever the **WfsAudioProcessor** detects that an update is needed, it sets the corresponding update flag. The update routine is handled in either an asynchronous timer, or in a place where it cannot affect the audio processing.

This is a list of the update flags:

- redrawLoudspeakers
- redrawXyPad
- updateLoudspeakerArray

6 Binaural showcase

Due to the ongoing Corona-pandamic we did not have the opportunity to test our plug-in with a physical array of loudspeakers. However binaural impulse responses of such an array had been recorded in a previous work. These impulse responses used a linear set-up with 24 loudspeakers, spaced by 15.1cm. An artificial dummy head was placed 2 meters away from the setup and for each loudspeaker and ear an impulse response was recorded.

Figure 10: Setup for recording binaural impulse responses

Two binaural example recordings have been made and uploaded as videos:

- speech demo
- musical demo

In order to use the plugin in its binaural version, the user needs a plugin that can convolve these 48 impulse responses in real time. We recommend Matthias Kronlachner's mcfx convolver. The IRs and an additional guide on how to use them can be found in the same repository as this documentation.

7 Conclusion

Our work implemented and tested a real-time wave-field synthesis plugin. It is based on a 2.5D approach, where the electroacoustic sources use 3 dimensions, but the final placement of sources and virtual sources only uses 2 dimensions.
The wave-field synthesis consists of 3 building blocks per loudspeaker: filter, delay and gain. The filter used in our implementation was a finite impulse response to easily accommodate the 3dB per octave slope. For each loudspeaker the filter is the same, thus the input audio signal only needs be filtered once. The delay block is implemented via a delay line using a circular buffer. The gain is evaluated and applied for each loudspeaker using its coordinates as well as the current source and listener positions.
The plugin is implemented in C++ using the JUCE framework. It offers a GUI where all setup and wave-field synthesis parameters can be controlled by the user. Controls can

also be automated, or sent and received over OSC. To ensure real-time safety the plugin decouples resource-consuming tasks from the audio processing.
Due to the access restrictions in the ongoing Covid-19 pandemic the plugin was only tested using a binaural rendering plugin and impulse responses from previous work. The instructions how and where to obtain the plugin and its binaural test environment can be found in this paper and in the respective code repository[2].

[2]https://git.iem.at/audioplugins/iem-wfs/

References

[1] G. Firtha, *A Generalized Wave Field Synthesis Frameworkwith Application for Moving Virtual Sources*. PhD thesis, Budapest University of Technology and Economics, 4 2019.

[2] E. Corteel and R. Pellegrini, "Wave field synthesis with increased aliasing frequency," vol. 1, 08 2008.

[3] L. Gölles and L. Maier, *Wellenfeldsynthese*. PhD thesis, University of Tecchnology Graz, 6 2020.

[4] F. Schultz, V. Erbes, S. Spors, and S. Weinzierl, "Derivation of iir prefilters for soundfield synthesis using linear secondary source distributions," pp. 2372–2375, 03 2013.