Renewed architecture of the *sWONDER* software for Wave Field Synthesis on large scale systems

Marije A.J. Baalman and **Torben Hohn** and **Simon Schampijer** and **Thilo Koch**
Institute for Audio Communication (Sekr. EN8)
Einsteinufer 17
10587 Berlin
Germany
baalman@kgw.tu-berlin.de and torbenh@gmx.de and simon@schampijer.de and tiko@admin-box.com

Abstract

For large Wave Field Synthesis (WFS) systems multiple computers are needed for rendering to manage the necessary amount of audio channels. To make this possible with the *sWONDER* software, the software was completely restructured and divided into several separate programs which can run on multiple computers, communicating with each other via OpenSoundControl. This paper describes the new structure of the program, as well as several implementation details of the scheduling unit and audio rendering unit.

Keywords

Auralisation, Wave Field Synthesis, Convolution

1 Introduction

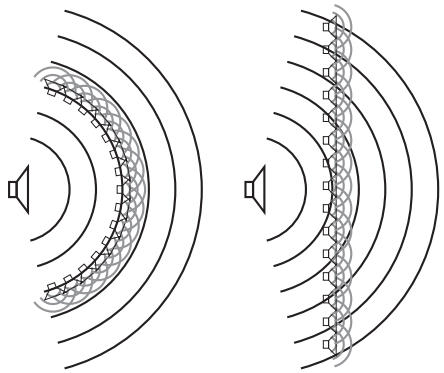
Wave Field Synthesis (WFS) is a method for sound spatialisation. Its main advantage is that it has no sweet spot, but instead a large listening area, making the technology attractive for concert situations.

The main principle of WFS is illustrated in figure 1. A wave field can be synthesized by a superposition of wave fields caused by a lot of small secondary sources, provided you calculate the right delays and amplitude factors for the source signal for each secondary source.

For large WFS systems the calculation of the audio signals for each loudspeaker cannot be done on just one computer, due to limitations of the CPU-power and hardware considerations, such as the number of output channels. Thus, a cluster of computers is necessary, and there is a need to synchronise these calculations. The previous versions of *sWONDER* [1; 2] were monolithic programs, which did not provide this option. This paper describes a new structure for the *sWONDER* program, which enables the software to control large scale WFS systems.

2 Hardware setup

In 2006/2007, the TU Berlin launched a project to equip one of the lecture halls with a large



(a) The Huygens' Principle (b) Wave Field Synthesis

Figure 1: From the Huygen's Principle to Wave Field Synthesis

WFS system[3; 4], of in total 840 loudspeaker channels, both for sound reinforcement during the regular lectures, as well as to have a large scale WFS system for both scientific and artistic research purposes. The loudspeakers are built into loudspeaker panels[5], each providing 8 audio channels, which are fed with an ADAT signal. Each panel additionally has 2 larger speakers which emit the low-pass filtered sum of the 4 channels above it.

To drive these speakers a cluster of 15 Linux computers is used. Each computer computes the loudspeaker signals for 56 loudspeaker channels. Each computer is equipped with an RME HDSP MADI[6] sound card. Each MADI output is connected to an MADI to ADAT bridge (RME ADI648[6]), which is mounted inside the wall, so that the ADAT cables can be kept short (up to 10 meters). The input to the system is multiplexed to each MADI sound card with the use of MADI bridges (RME MADI Bridge[6]).

The cluster has two networks, one for the OSC[7] communication, and one for data-transfer. Separating these network functions,

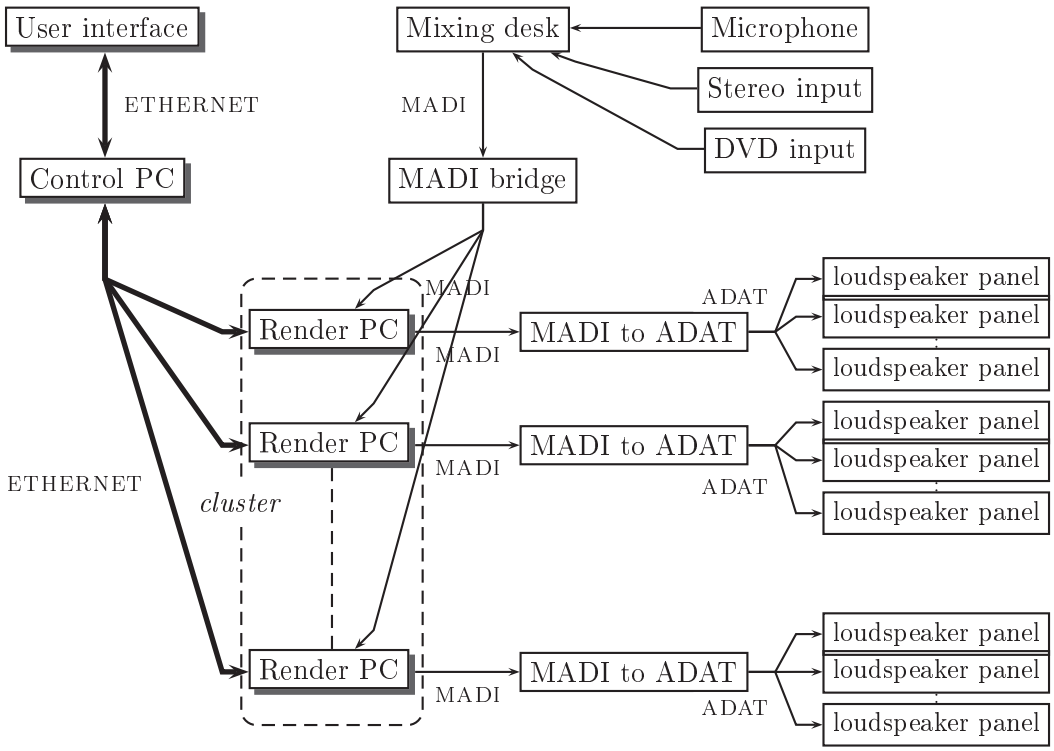


Figure 2: Schematic overview of the hardware setup for the WFS system in the lecture hall of the TU Berlin.

ensures that the OSC communication is fast. The master machine (Control PC) acts as a bridge to the outside world and is the only computer that is connected to an external network.

A general overview of the hardware setup is given in figure 2.

The *sWONDER* software was adapted to control this system, in such a way, that it can also be used by similar but not necessarily identical systems.

3 Software architecture

The software is divided in several parts:

- a graphical user interface,
- a score player/recorder,
- a control unit,
- a real-time render unit,
- an offline render unit
- and a common library for general functions.

Communication between the different parts of the program is based on the OSC protocol[7]. Figure 3 gives an overview of the program parts and their communication.

3.1 Graphical User Interface

The graphical user interface (GUI) provides dialogs for loudspeaker array configuration, grid point configuration (possible source positions and their characteristics), composition and a real time control interface. In the real time control interface, it is possible to move sources around with the mouse, as well as to store different scenes, between which can be switched. The GUI is currently still in development, and will be based on the current GUI [1; 2]. It will be ported to *Qt4* [8], its usability will be improved, and we are working on ways to visualise the timeline of two-dimensional movement.

3.2 Score player/recorder

The system can take any kind of audio input, so that the user can use the audio player (s)he prefers to play the audio. The score player/recorder is used to synchronise with an audio player and record and playback source movements. Synchronisation is based on MTC (Midi Time Code), as this is a clock format which many DAW's support.

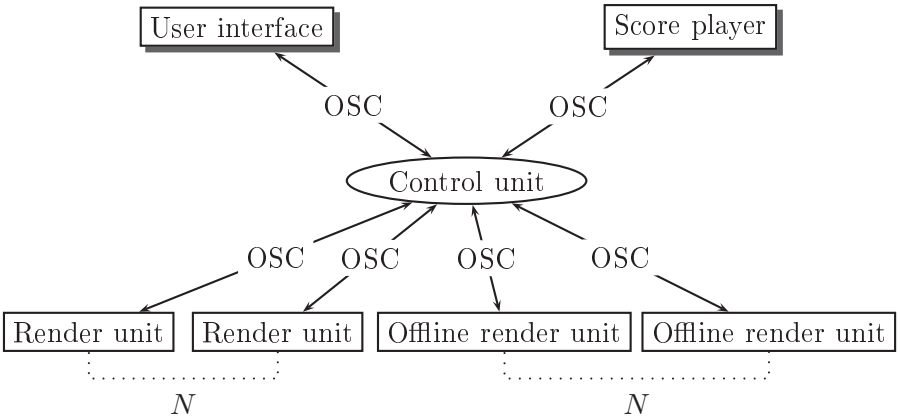


Figure 3: Schematic overview of the different parts of the software *sWONDER*. The control unit can communicate with an arbitrary number (N) of realtime and offline renderers.

3.3 Offline renderer

For room simulations or for complex sound sources [9], the calculations for the impulse responses for each speaker can take quite long, and cannot be performed in realtime. For this purpose, there will be an offline render unit, which takes care of all these calculations, utilising the benefit from parallel execution on a cluster.

3.4 Control unit

The control unit acts as a bridge between the user interface and the audio renderers; it also communicates with the score player/recorder. Though the *sWONDER* suite of programs will also supply a graphical user interface, any other program that can send (and receive) OSC can be used to control the system. The user interface only needs to communicate with the control unit, and does not need to know anything about the audio rendering details; the control unit takes care of that.

3.5 Rendering engine

The real-time render unit is responsible for the actual audio signal processing. It has several ways to deal with the audio streams: playback of direct sound, utilising weighted delay lines, convolution of the input sound for early reflections, and convolution of the input sound for reverb followed by a weighted delay lines to create plane waves with the reverb tail. Schematically this is shown in figure 4.

The rendering engine consists of two parts: *twonder* for the delay line implementation, and *fwonder* for the convolution. Both programs are controlled by OSC; audio input and output has JACK as the audio backend.

4 Direct sound

4.1 Delay lines

The direct sound of a WFS synthesized source, consists of the delayed and attenuated source signal. This delay and attenuation is unique for each speaker. The direct sound of the source is rendered in the time-domain by the *twonder* part of the program.

To initialise the delay lines, the length of the delay lines need to be determined. The length is related to the largest distance a source will have to a speaker. Also, it needs to be decided how far in front of the speakers we want to move a source, as this determines the needed delay offset. If no focused sources are needed, we can set the delay offset to a smaller number, thus introducing less latency in the system. These options can be set per source.

4.2 Moving sources

When a source moves, the delay time will change continuously, as well as the volume factor. In *twonder* the delay time for the start and the end of the block is calculated (thus these are a kind of anchor points), and the samples inside the block are resampled. This is clarified in figure 5. If the delay time is 20 samples at the start of the block, and 30 samples at the end of the block, we need to output 10 samples less than the actual block size N. Thus, we need to resample $N - 10$ to N samples. Because of the CPU restraints (we need to do this for a lot of delaylines in realtime), we need an efficient resampling algorithm. We chose linear interpolated resampling. The implementation is a modified version of Bresenham's line drawing

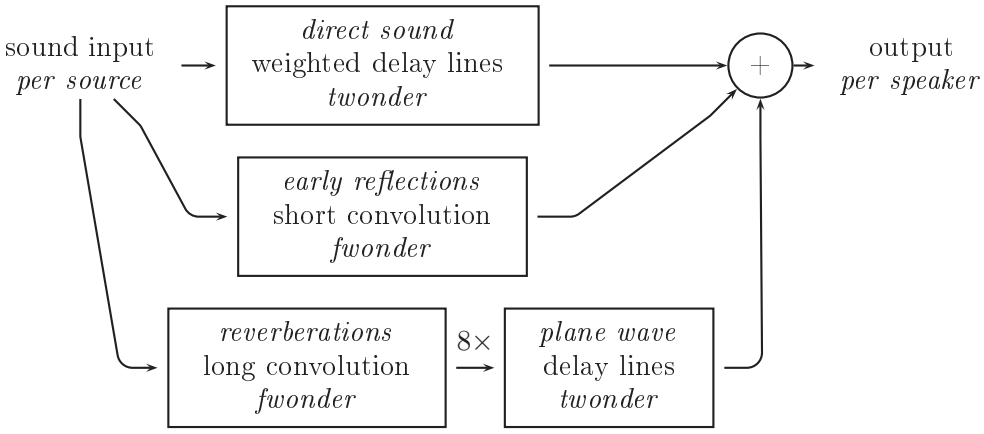


Figure 4: An overview of the audio signal processing by the real-time render unit

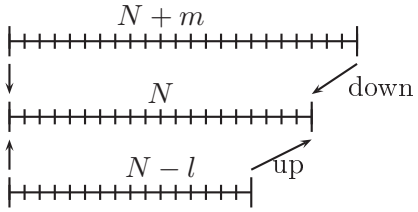


Figure 5: Illustration of the resampling problem: if the delay time gets longer within a certain block, we need to output more samples than we have available in our buffer. Thus, we need to upsample the avaible samples. If the delay time gets shorter, we need to output less samples, than we have available in our buffer and we need to downsample them.

algorithm [10], which eliminates the need to cast a float to an integer in the innner loop.

Moving a source in this way, creates a Doppler effect, which will be audible if the movement is very fast. In some cases it is not desired to hear a Doppler effect, so another option for movement is provided, which we have called a *fade jump*. Using this option, the illusion of movement is created by fading the source out on one position, while fading it in on the next position. The update frequency for this can be set by the user.

4.3 Plane waves

Plane waves are achieved by just varying the delay times for each speaker, based on the angle the wave front makes with the speaker array. A delay offset is created by giving the plane wave a point of origin in space, in addition to its direction. This approach also makes it possible to switch from a point source to a plane wave and vice versa.

Plane waves can be used to simulate sources that are very far away and only have a direction, or to simulate reflections, as will described in the next section.

5 Room simulation

Room simulation is achieved by adding reflections to the direct sound. This can be achieved in several ways: (1) inclusion of a first reflection in the delay line, (2) doing a short convolution for early reflections for each speaker with offline calculated impulse responses (IRs) and (3) doing a convolution with a longer impulse response, the result of which will be played back using plane waves.

The first option is in development. In this option also a filter on the reflected sound can be included, provided the filter can be created with ca. 8 FIR taps.

The second and third option are possible already, though the offline renderer to calculate the early reflection impulse responses is not ready yet. Alternately, other methods could be used to calculate the early reflection IRs, such as an old version of *sWONDER*, or using an approach based on measurements such as described in [11; 12; 13]

Ad 2: The impulse responses are unique to each source position and speaker. Thus for each speaker a convolution needs to be made. This option is CPU-intensive, and requires all of the impulse responses to be loaded into memory. In [14] research is presented from which can be concluded how closely gridpoints need to be spaced to ensure perceptual consistency of the wave field, for a specific setup (depending on the dimensions of the virtual room, as well as the size

of the desired listening area).

Ad 3: research at the TU Delft has proved that using 8 plane waves (at 45 degrees interval directions) is sufficient to create a realistic reverberation[15].

5.1 Convolution

The *fwonder* program implements a fast convolution from multiple inputs to (even more) multiple outputs. It uses the same complex multiplication method as BruteFIR [16]. Instead of extending BruteFIR we rewrote a convolution engine from scratch, because this was considered faster than extending BruteFIR, due to the lack of transparency and documentation of the BruteFIR code. The other available solutions were not written in C++ or tied to SuperCollider [17; 18], which would have slowed down development also. So we decided to reimplement the algorithm, while learning from the others.

5.2 IR caching

When a source is moving, we need to change the impulse responses being used. Because the set of impulse responses does not fit into memory, a cache structure needs to manage the loading of the impulse responses from disk.

This problem is solved as follows: when the position of a source changes the UI sends the absolute position in meter to the control unit. The control unit sends the new position to *twonder*, and simultaneously calculates the corresponding (closest) grid position for which an early reflection impulse response is available, and sends this information to the render unit. The render unit then switches the impulse responses used in the convolution to the new ones. Crossfading is used to reduce the artefacts of this process.

Because the loading of new impulse responses is a task that takes some time to complete, it should happen before an event actually occurs if possible. In real-time mode we do not know in advance what parameters of which source will change next. As a solution the grid of points for which IRs are calculated is divided in anchor points and normal points. Anchor points are points whose IRs are always stored in memory. When a source moves to a new location, first the IR of the anchor point is used, and then the surrounding points are loaded into memory, so that changes to locations nearby can be made in real-time (see figure 6). When there is a score, we do know which IRs are needed in the future, and we can determine the needed IRs in time, as shown in figure 7.

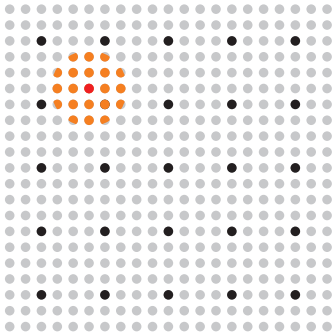


Figure 6: Loading grid point impulse responses into cache. The black points are the anchor points and correspond to impulse responses that are always loaded in memory. The red (darkest grey) point indicates the grid point used for the current position, the orange (grey) points the one for which the IRs are currently loaded in memory. The light grey points are the available points.

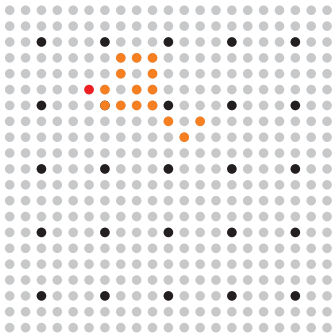


Figure 7: Loading grid point impulse responses into cache while playing a score. As we know the future locations of the source, we can preload the IRs that correspond precisely to the sound path.

The control unit takes care of this scheduling of loading and unloading of IRs and sends commands to the render units to perform this (i.e. the render unit is 'stupid' and just follows the orders of the control unit).

5.3 Calculating the IRs

The IRs as described above, will need to be calculated beforehand with the offline renderer. This is handled as follows: in the UI the user

defines a grid of points and virtual room dimensions. Then he sends a message to the control unit to start the calculation. The control unit then communicates with all the offline renderers that are running, to perform this task, and sends a message back to the UI when the task is completed. Then the calculated IRs can be used in realtime.

6 Time and synchronisation

There are several concepts of time within the system: the user interface can send messages, which have to be executed *now* and have a certain *duration*; or it can send messages which have to be executed at a certain time from *now* and have a certain *duration*.

The score player/recorder has to deal with both MTC and synchronise itself to that clock, as well as communicate to the control unit, just like other user interfaces.

All communication from the user interface to the control unit about time, is in seconds. As the renderers need to be synchronised with sample accuracy, the control unit translates the time in seconds to frame time. The audio clock is used as the time reference. This clock is reliable, has got the desired granularity and is present on each render unit and the control unit. The audio devices in the units are fed with a MADI signal including a word clock signal. Because the audio links are digital, a sawtooth generated at the control node, will be sufficient to extract the initial synchronisation position from the audio signal. When initial synchronisation is done, sync will be maintained by the word clock sync.

This leads to a system with one central clock and avoids the need for clock skew compensation which is needed when having multiple clocks.

As an example we consider the task of changing the position of a source. This information is sent from the UI to the control unit where a timestamp for this event is generated. Since the control unit has the information about the actual time in samples the messages will be stamped with this time reference and send to the render unit.

Both the control unit and the render unit can deal with interpolation over time, i.e. it is possible to send the control unit a message to move a source from one position to another with a certain duration of the movement. The control unit will pass on this duration to *twonder*, which then interpolates the movement and calculates the positions (and thus the delays) at the end of each block, and creates the movement. The control unit will also calculate the intermediate positions on the grid, and ensure that the IRs for the intermediate points are preloaded by *fwonder* and the IRs needed for the current position are switched to in time.

7 File formats

For configuration of the system and creating a project with the system, several files are needed to store the relevant data.

It was chosen to use XML for the format for storing this data, as it is easily extendible in case of need.

There are files for:

Configuration This contains the data about the rendering units: the network setup and the speaker setup.

Project This contains the general settings for a project, such as how many sources are used and the characteristics of each sources. It can also contain a score, and settings for different scenes (static constellations of sources, between which the user can switch).

Grid This contains the information about the grid points used for early reflection calculation, as well as information about the impulse responses (path and format in which they are stored).

As a basis for the project file format we used the XML-format for 3D audio as described in [19]. Currently we are undertaking efforts to start a discussion with other institutes that work on Wave Field Synthesis to agree upon a common XML-format to be able to exchange content.

8 Working OSC commands

In table 1 an overview is given of the currently working OSC commands.

8.1 Project

To be able to store scenes, you need to create a new project with the command: `/WONDER/project/create`, with one *string* as argument: the project name.

You can save the project with the command: `/WONDER/project/save`, and later load it again with the command `/WONDER/project/load`.

command	types	arguments
/WONDER/project/create	s	projectname
/WONDER/project/load	s	projectname
/WONDER/project/save	s	projectname
/WONDER/scene/add	i	scene no.
/WONDER/scene/select	iff	scene no., time, duration
/WONDER/scene/remove	i	scene no.
/WONDER/scene/set	i	scene no.
/WONDER/source/position	iffffff	srcid, pos x, pos y, pos z, time, duration
/WONDER/source/angle	ifff	srcid, angle, time, duration
/WONDER/source/type	iiff	srcid, type, angle, time

Table 1: Working OSC commands

8.2 Scenes

You can create a snapshot of the current source positions (called a "scene") and store them in the project, using the command `/WONDER/scene/add` with an integer as argument for the slot number under which you want to store the scene.

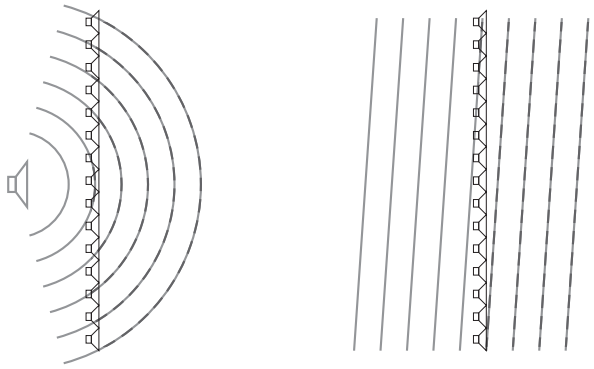
Later you can recall the scene with the command `/WONDER/scene/select`, with as arguments the scene number, the time at which the change to the scene should start, and the duration in which it should fade to the new scene.

With `/WONDER/scene/remove` a scene is deleted (and thus the slot is freed again). With `/WONDER/scene/set` you can overwrite an existing scene. Note the subtle difference between adding a scene and setting a scene: adding creates a new scene and stores the current source positions to it. It gives an error back when the scene number already exists. "Set" stores the current source positions to an existing scene and gives an error back if the scene slot does not exist.

8.3 Source control

There are two types of sources: point source (see fig. 8a) and plane wave (see fig. 8b).

With the command: `/WONDER/source/type` you can set the type for one source. Plane wave is "0", point source is "1". The angle argument is the start angle for the plane wave. Whenever the type is changed you should also send a `/WONDER/source/position` command, to set the position of the source. In the case of a point source, this will be the actual position of the source. In the case of a plane wave, this is a reference point for the calculation; it should be chosen to be a position somewhere behind the array in the direction where the plane wave is coming from. This point determines the basic latency of the plane wave.



(a) Point source (b) Plane wave

Figure 8: Source types

`/WONDER/source/position` takes as arguments the source id, the x and y position (in meters), the z position (which should be 1.0 for now), the time at which the change should start (in seconds from "now"), and the duration for the change to take place (also in seconds).

`/WONDER/source/angle` takes as arguments the source id, the angle, the time at which the change should start, and the duration for the change to take place.

9 Conclusions

We have presented the new architecture of the *sWONDER* software, with a focus on the central control unit and the audio rendering unit. The user interface of *sWONDER*, a score player and offline render unit are in development, to provide a full suite of open source tools for doing WFS.

Parts of the software may also be useful for other purposes, such as the OSC-controllable de-

laylines and convolution engine.

We plan to extend the software with options for other spatial reproduction techniques, such as binaural headphone reproduction and ambisonics.

10 Acknowledgements

Our thanks go to the Bauabteilung of the TU Berlin for funding, and especially to Christoph Moldrzyk for initiating the project.

The software is released under the GPL license at `http://swonder.sourceforge.net`.

References

[1] M.A.J. Baalman. Application of wave field synthesis in electronic music and sound installations. In *2nd International Linux Audio Conference, 29 april - 2 Mai 2004, ZKM, Karlsruhe*, 2004.

[2] M.A.J. Baalman. Updates of the wonder software interface for using wave field synthesis. In *3rd International Linux Audio Conference, April 21-24, 2005, ZKM, Karlsruhe*, 2005.

[3] C. Moldrzyk, A. Goertz, M. Makarski, W. Ahnert, S. Feistel, and S. Weinzierl. Wellenfeldsynthese für einen großen hörsaal. In *DAGA 2007, Stuttgart, Germany*, 2007.

[4] T. Behrens, W. Ahnert, and C. Moldrzyk. Raumakustische konzeption von wiedergaberäumen für wellenfeldsynthese am beispiel eines hörsaals der tu berlin. In *DAGA 2007, Stuttgart, Germany*, 2007.

[5] A. Goertz, M. Makarski, C. Moldrzyk, and S. Weinzierl. Entwicklung eines achtkanaligen lautsprechermoduls für die wellenfeldsynthese. In *DAGA 2007, Stuttgart, Germany*, 2007.

[6] Rme - intelligent audio solutions. `http://www.rme-audio.com/`.

[7] M. Wright, A. Freed, and A. Momeni. Opensoundcontrol: State of the art 2003. In *2003 International Conference on New Interfaces for Musical Expression, McGill University, Montreal, Canada 22-24 May 2003, Proceedings*, pages 153–160, 2003.

[8] Trolltech. Qt library. `http://www.trolltech.com/products/qt/index.html`, 1996-2005.

[9] M.A.J. Baalman. swonder3dq: Auralisation of 3d objects with wave field synthesis. In *4th International Linux Audio Conference, April 27-30, 2006, ZKM, Karlsruhe*, 2006.

[10] Wikipedia. Bresenhams line algorithm. `http://en.wikipedia.org/wiki/Bresenham's_line_algorithm`, 2007.

[11] Frank Melchior and Diemer de Vries. Detection and visualization of early reflections for wave field synthesis sound design applications. In *Tonmeistertagung 2006, Leipzip, Germany*, November 16-19 2006.

[12] Diemer de Vries, Jan Langhammer, and Frank Melchior. A new approach for direct interaction with graphical representations of room impulse responses for the use in wave field synthesis reproduction. In *120th AES Convention, Paris, Preprint 6657*, May 2006.

[13] Edo Hulsebos and Diemer de Vries. Spatial decomposition and data reduction of sound fields measured using microphone array technology. In *17th ICA, Rome, 2001*, 2001.

[14] Hiske Helleman. Sensitivity of the human auditory system to spatial variations in single early reflections. Master's thesis, Delft University of Technology, The Netherlands, 2003.

[15] J.-J.Sonke and D. de Vries. Generation of diffuse reverberation by plane wave synthesis. In *102nd AES Convention, March 1997, Preprint 4455*, 1997.

[16] A. Torger. Brutefir. `http://www.ludd.luth.se/~torger/brutefir.html`, 2001-2005.

[17] J. McCartney. Supercollider. `http://www.audiosynth.com`.

[18] Stefan Kersten. A fast convolution engine for the virtual electronic poem project. Master's thesis, Technische Universität Berlin, 2006.

[19] Guillaume Potard. *3D-Audio Object Oriented Coding*. PhD thesis, University of Wollongong, Australia, September 2006.