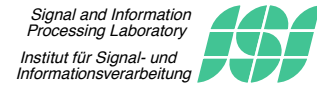




Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich



Spring Semester 2010

Prof. Hans-Andrea Lölicher

Semester Thesis

An Audio Engine for an Ambisonics Surround Sequencer

June 16, 2010

Samuel Gähwiler

Advisor: Dr. Kurt Heutschi

Co-Advisor: Philippe Kocher

Project Description

Introduction

Ambisonics is a series of recording and replay techniques using multichannel mixing technology that can be used live or in the studio. By encoding and decoding sound information on a number of channels, a 2-dimensional (“planar”, or horizontal-only) or 3-dimensional (“periphonic”, or full-sphere) sound field can be presented. (Wikipedia)

Even though invented in the early 1970s, it has never been well marketed and is therefore not widely known. For musicians, composers and artists it is quite difficult to work with ambisonics. This project aims to simplify this.

Tasks

Your assignment is to design software capable of playing back multiple tracks of audio. You also make yourself familiar with the surround-format ambisonics by reading papers. A next step is to design and implement a flexible and configurable ambisonics decoder.

General Regulations

At the end of your project, you will have to hand in a report (an original and two copies, all signed) which is typeset in \LaTeX . The original as well as the copies are property of the laboratory. Be prepared to produce a CD containing all the source code and important parts of data that you have written or generated.

Before the end of your project, you will have to give a 15 minutes presentation on your project. The exact date will be announced.

There will be a mandatory introduction to the lab and its facilities where further details and regulations are going to be explained.

Dates

Start : Monday, 15 March 2010

End : Wednesday, 16 June 2010, 12:00 PM

Addresses

Prof. H.-A. Loeliger

ETF E 101

<loeliger@isi.ee.ethz.ch>

+41 44 632 27 65

Dr. K. Heutschi

ETF D 109.2

<kurt.heutschi@isi.ee.ethz.ch>

+41 44 632 77 08

P. Kocher

ICST

<philippe.kocher@zhdk.ch>

Acknowledgments

I'd like to thank Kurt Heutschi and Hans-Andrea Loeliger for making this cooperation with the ICST possible at all. Also many thanks to Philippe Kocher and the other nice people at the ICST, for giving me the opportunity to work on such an interesting topic.

Samuel Gaehwiler

Abstract

A brief introduction to Ambisonics and Ambisonics Equivalent Panning (AEP) is given, followed by the description of the signal processing part of the Choreographer software, a tool to place audio recordings as virtual sound sources into space. The Choreographer feeds an array of speakers which simulate the sound field of these virtual sources.

Contents

1	Introduction	1
2	A Short Introduction To Ambisonics	3
2.1	Ambisonics in 2 Dimensions	3
2.2	Ambisonics in 3 Dimensions	6
2.3	Ambisonics Equivalent Panning (AEP)	7
3	Code Documentation	8
3.1	GUI to Audio Engine Communication	8
3.2	The Transfer of Digital Audio to the DAC	9
3.2.1	Multiple callbacks	10
3.2.2	Callbacks in Juce	10
3.3	Structure of the Audio Engine	11
3.3.1	AmbisonicsAudioEngine	11
3.3.2	AudioSourcePlayer	12
3.3.3	AudioTransportSource	12
3.3.4	AudioSampleBuffer	12
3.3.5	AudioRegionMixer	12
3.3.6	AudioSourceAmbipanning	13
3.3.7	AudioSourceGainEnvelope	13
3.3.8	ResamplingAudioSource	14
3.3.9	AudioFormatReader	14
4	Future Work	15
4.1	Improvements	15
4.2	Additional Features	15
A	Terminology	16

Bibliography**16**

CONTENTS

SEMESTER THESIS

Chapter 1

Introduction

This project deals with the signal processing in the software Choreographer. The creation of this computer program has been initiated by Philippe Kocher, a member of the Institute for Computer Music and Sound Technology (ICST) at the Zürcher Hochschule der Künste. The signal processing part is written in C++ and uses JUCE [1], a class library for developing cross-platform applications, which is quite popular in the music software industry.

In order to understand this project, some knowledge of Ambisonics and AEP is needed, which is provided in Chapter 2. In Chapter 3, the basic structure of the code and the tasks of the classes in use are presented.

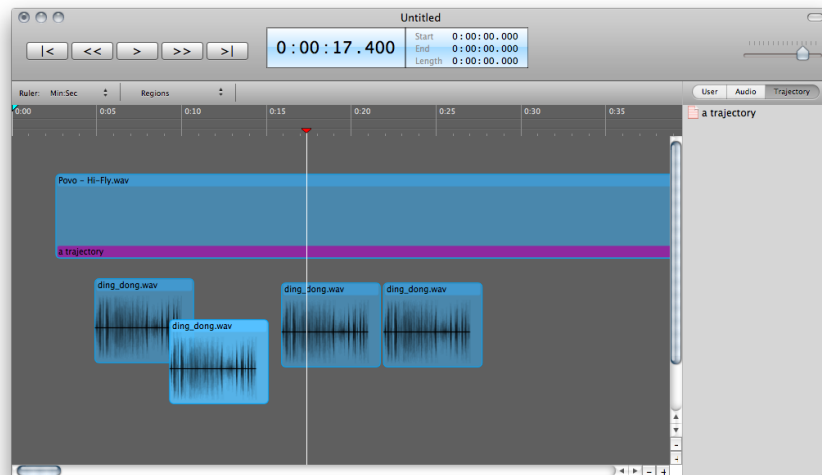


Figure 1.1: The arranger window of the Choreographer.

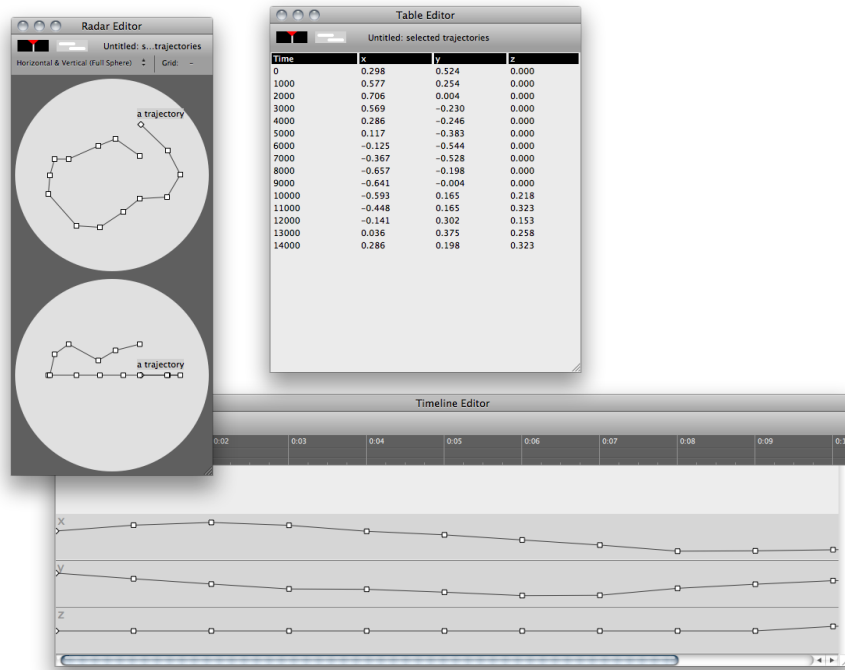


Figure 1.2: The radar, table and timeline editors of the Choreographer. They are used to alter a spacialisation envelopes (trajectories).

Chapter 2

A Short Introduction To Ambisonics

This section is an attempt to provide a rough idea of what Ambisonics and Ambisonics Equivalent Panning are. It is based on publications by Martin Neukom [2], [3]. This is not a derivation, the curious reader refers to [4].

2.1 Ambisonics in 2 Dimensions

Lets start in two dimensions, since it is easier to understand and still shares the same ideas that are also applied in three dimensions.

Consider the setup, where the listener is positioned exactly in the center of a unit circle. N speakers are placed at arbitrary positions on this circle, each pointing to the center of the circle. We would like them to produce a sound field that is indistinguishable of the sound field of an arbitrary virtual source, at least for the listener at the central position. It is assumed that the virtual source is also located on the unit circle and pointing to the center. A further assumption is that all sources emit plane waves. This configuration is pictured in Figure 2.1. For $n \in \{1, 2, \dots, N\}$ the signal of the n -th speaker is denoted by p_n and its position by the angle ψ_n . s is the signal of the virtual source and its position is given by the angle ψ . s and all the ψ 's are given and we would like to figure out p_1, p_2, \dots, p_N . $\psi_1, \psi_2, \dots, \psi_N$ are fixed values, $s, \psi, p_1, p_2, \dots, p_N$ are functions of time.

To extend s to a function of time and position (angle φ), it can be

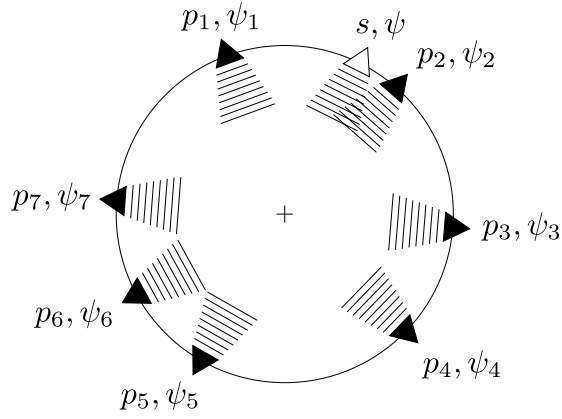


Figure 2.1: Configuration

multiplied with an indicator function

$$s \cdot I(\varphi - \psi).$$

Using the Fourier series expansion, it can be approximated by

$$s \cdot I(\varphi - \psi) \approx s \cdot \frac{1}{M+1} \sum_{m=0}^M \cos(m(\varphi - \psi)), \quad (2.1)$$

where M is the order of the approximation. The left side of (2.1) is visualised in Figure 2.2, the right side in Figure 2.3.

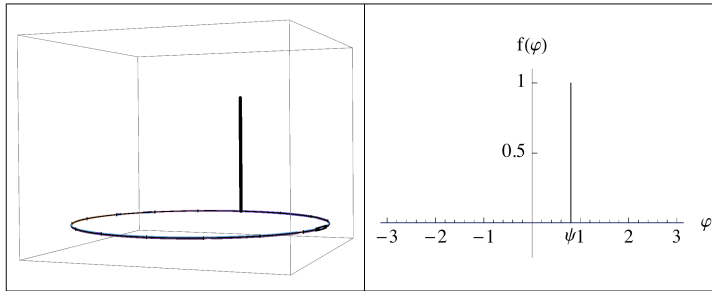


Figure 2.2: Indicator function

The same can be done with the signals of the speakers. Since the sum of all speakers should yield the same sound field as the sound field of the virtual source, we can write

$$s \cdot \frac{1}{M+1} \sum_{m=0}^M \cos(m(\varphi - \psi)) = \sum_{n=1}^N \left(p_n \cdot \frac{1}{M+1} \sum_{m=0}^M \cos(m(\varphi - \psi_n)) \right).$$

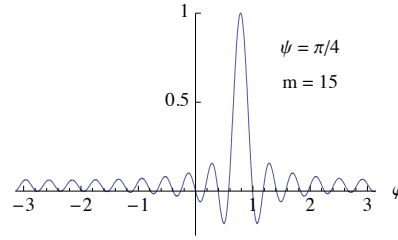


Figure 2.3: Approximation of the indicator function

Using the fact that $\{\sin(k\varphi), \cos(k\varphi); k \in \mathbb{N}\}$ is a set of linear independent functions and

$$\cos(\alpha - \beta) = \cos(\alpha) \cos(\beta) + \sin(\alpha) \sin(\beta)$$

yields $2M + 1$ equations:

$$\begin{aligned} s &= \sum_{n=1}^N p_n \\ s \cos(m\psi) &= \sum_{n=1}^N p_n \cos(m\psi_n), \quad m \in \{1, 2, \dots, M\} \\ s \sin(m\psi) &= \sum_{n=1}^N p_n \sin(m\psi_n), \quad m \in \{1, 2, \dots, M\}. \end{aligned}$$

They can be written in matrix notation

$$\underbrace{\begin{pmatrix} 1 \cdot s \\ \cos(\psi) \cdot s \\ \sin(\psi) \cdot s \\ \cos(2\psi) \cdot s \\ \sin(2\psi) \cdot s \\ \vdots \\ \cos(M\psi) \cdot s \\ \sin(M\psi) \cdot s \end{pmatrix}}_{\mathbf{b}} = \underbrace{\begin{pmatrix} 1 & 1 & \dots & 1 \\ \cos(\psi_1) & \cos(\psi_2) & \dots & \cos(\psi_N) \\ \sin(\psi_1) & \sin(\psi_2) & \dots & \sin(\psi_N) \\ \cos(2\psi_1) & \cos(2\psi_2) & \dots & \cos(2\psi_N) \\ \sin(2\psi_1) & \sin(2\psi_2) & \dots & \sin(2\psi_N) \\ \vdots & \vdots & & \vdots \\ \cos(M\psi_1) & \cos(M\psi_2) & \dots & \cos(M\psi_N) \\ \sin(M\psi_1) & \sin(M\psi_2) & \dots & \sin(M\psi_N) \end{pmatrix}}_{\mathbf{A}} \cdot \begin{pmatrix} p_1 \\ p_2 \\ \vdots \\ p_N \end{pmatrix}. \quad (2.2)$$

The left side of equation (2.2) could also be replaced by $\mathbf{b}_1 + \mathbf{b}_2 + \dots + \mathbf{b}_K$, if there were K virtual sources to be simulated. The signal vector on the left side is called the *Ambisonics B-Format*. The dimension of this vector ($2M + 1$) is independent of the number of speakers N .

The $(2M + 1) \times N$ matrix is fully determined by the fixed positions (angles) of the speakers. To figure out p_1, p_2, \dots, p_N , we have to solve the equation system (2.2). For once it might be a good idea to calculate the inverse or the pseudo-inverse $(A^T \cdot (A \cdot A^T)^{-1})$ of A . If the speakers are evenly distributed and $N = 2M + 1$

$$\begin{pmatrix} p_1 \\ p_2 \\ \vdots \\ p_N \end{pmatrix} = \frac{1}{N} A^T \cdot \mathbf{b}. \quad (2.3)$$

2.2 Ambisonics in 3 Dimensions

In the three dimensional setup, the speakers are now placed on a unit sphere around the listener. The idea is quite similar, but the indicator functions on the surface of the sphere are approximated by the so called *spherical harmonics*. Similar to $\sin(\cdot)$ and $\cos(\cdot)$, the spherical harmonics are an orthogonal basis for functions from the sphere to \mathbb{C} . By performing similar steps as before, one also gets a linear system of equations, similar to (2.2).

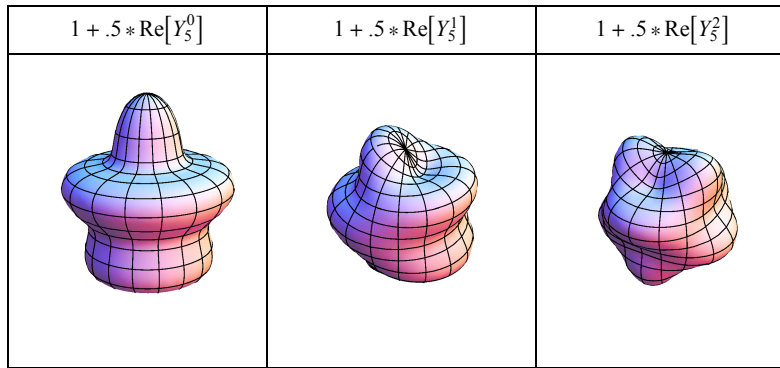


Figure 2.4: The spherical harmonics Y_5^0, Y_5^1 and Y_5^2 .

Remark: In contrast to the two dimensional case, where setting up the speakers at equidistant positions is possible for any number of speakers, on the unit sphere there are only 5 equidistant speaker setups possible. Placed in the corners of the 5 platonic bodies.

2.3 Ambisonics Equivalent Panning (AEP)

In the Choreographer, there is no need for the explicit calculation of the B-Format vector. Starting from (2.3) - equidistantly placed speakers are now assumed - we get

$$\begin{pmatrix} p_1 \\ p_2 \\ \vdots \\ p_N \end{pmatrix} = \frac{1}{N} A^T \cdot \mathbf{b} = \frac{1}{N} A^T \cdot \begin{pmatrix} 1 \\ \cos(\psi) \\ \sin(\psi) \\ \cos(2\psi) \\ \sin(2\psi) \\ \vdots \\ \cos(M\psi) \\ \sin(M\psi) \end{pmatrix} \cdot s = \begin{pmatrix} G_1(\psi) \\ G_2(\psi) \\ \vdots \\ G_N(\psi) \end{pmatrix} \cdot s. \quad (2.4)$$

In 2 dimensions, a factor for speaker $n \in \{1, 2, \dots, N\}$ is given by

$$G_{2D}(\gamma) = \frac{1}{N} \left(g_0 + 2 \sum_{m=1}^M g_m \cos(m\gamma) \right), \quad (2.5)$$

where γ is the angle between the speaker and the virtual source. In 3 dimensions, it is

$$G_{3D}(\gamma) = \frac{1}{N} \sum_{m=0}^M (2m+1) g_m P_m(\cos(m\gamma)), \quad (2.6)$$

where $P_m(\cdot)$ is the Legendre polynomial of order m . There is no simplification known for (2.6), but it can be approximated by (2.5), for which quite a drastic simplification exists:

$$\begin{aligned} G_{2D}(\gamma) &= \left(\frac{1}{2} + \frac{1}{2} \cos \gamma \right)^M \\ &= \left(\frac{1}{2} + \frac{1}{2} \langle \mathbf{pos}_{\text{speaker}}, \mathbf{pos}_{\text{source}} \rangle \right)^M \\ &= \left(\frac{2 + x_{\text{speaker}} x_{\text{source}} + y_{\text{speaker}} y_{\text{source}} + z_{\text{speaker}} z_{\text{source}}}{2} \right)^M \end{aligned} \quad (2.7)$$

Calculating the signals for the speakers according to (2.4) with (2.7) is called *Ambisonics Equivalent Panning (AEP)*. The Choreographer uses AEP.

Chapter 3

Code Documentation

3.1 GUI to Audio Engine Communication

Internally, the Choreographer software consists of two parts. One deals with the graphical user interface (GUI), keeps track of all audio regions and envelopes, is responsible for saving and loading the current state, provides the undo functionality and more. From now on, it will be referenced with the term *GUI*. The other part is responsible for the actual signal processing. With *Audio Engine*, this part of the code is meant. This documentation deals with the latter for the most part.

To keep things simple, communication between the two parts can only be initiated by the GUI. The interface for it is provided by `AudioEngine.h` and `AudioEngine.mm`. This is also the bridge for the programming languages ObjectiveC, in which the GUI is written in, and C++, the language used for the audio engine. Listing 3.1 shows the part of `AudioEngine.h` that specifies this interaction.

Listing 3.1: Excerpt of AudioEngine.h

```

// Transport

- (void)startAudio:(unsigned long)playbackLocation;
- (void)stopAudio;

- (void)setLoopStart:(unsigned long)start end:(unsigned long)end;

// Getter

- (BOOL)isPlaying;
- (unsigned long)playbackLocation;
- (unsigned int)sampleRate;
- (double)cpuUsage;

// Setter

- (void)setMasterVolume:(float)dbValue;

- (void)setAmbisonicsOrder:(short)order;
- (void)setdBUnit:(double)unit;

- (void)setUseHipassFilter:(BOOL)filter;
- (void)setUseDelay:(BOOL)delay;

// Schedule Playback

- (void)addAudioRegion:(id)audioRegion;
- (void)modifyAudioRegion:(id)audioRegion;
- (void)deleteAudioRegion:(id)audioRegion;
- (void)deleteAllAudioRegions;

- (void)setGainAutomation:(id)audioRegion; // private
- (void)setSpatialAutomation:(id)audioRegion; // private

```

3.2 The Transfer of Digital Audio to the DAC

In order to make sure that the digital to analog converter (DAC) of the audio interface gets an uninterrupted stream of discrete time signals, the audio driver asks its client, e.g. the Choreographer, at regular intervals to

put a certain number of samples to a determined location in memory. Since this memory is allocated by the audio driver, it is ensured that there is always data present for the DAC to read out, even if the client was not able to finish filling the given memory location with its own samples in the time interval at disposal. Of course this case should never occur if there is enough processing power at hand and the client was carefully designed.

Part of this described mechanism is realised by a so called *callback function*, a function of the client, that is periodically called by the audio driver. Listing 3.2 shows the declaration of such a callback function.

Listing 3.2: an audio callback

```
audioDeviceIOCallback (const float** inputChannelData ,
                      int totalNumInputChannels ,
                      float** outputChannelData ,
                      int totalNumOutputChannels ,
                      int numSamples );
```

3.2.1 Multiple callbacks

It is common to split a big project into smaller blocks of code (or hardware), and this audio engine is no exception. Different objects manipulate the digital audio before it is ready for the audio interface. The same concept of callbacks is also used here to pass audio from one object to another. If possible, they even operate on the same memory location. E.g. an instance of **AudioSourceGainEnvelope** will directly forward the block of memory given as an argument to its own callback method to an instance of **ResamplingAudioSource**. After the **ResamplingAudioSource** instance has filled this block of memory, the **AudioSourceGainEnvelope** instance will alter the samples without changing their memory location (Section 3.3 helps to understand the underlying structure).

3.2.2 Callbacks in Juce

Listing 3.2 is actually taken from the **AudioSourcePlayer** class of the JUCE class library and is very close to the callback, the audio driver is calling. Most audio callbacks in the audio engine look a little different, though, as listing 3.3 reveals. The **AudioSourceChannelInfo** struct contains a pointer to a **buffer**-object, an integer that describes the start position of the block in the buffer and an integer that specifies the length of the block. The

main difference to the callback in Listing 3.2 - besides the packaging - is the fact that there are not two (input and output), but only one memory location given. This contains the input audio signal and (after reading it out if interested - not necessary in this audio engine) has to be overwritten with the output audio signal.

Listing 3.3: a slightly different audio callback

```
getNextAudioBlock (const AudioSourceChannelInfo& bufferToFill);
```

3.3 Structure of the Audio Engine

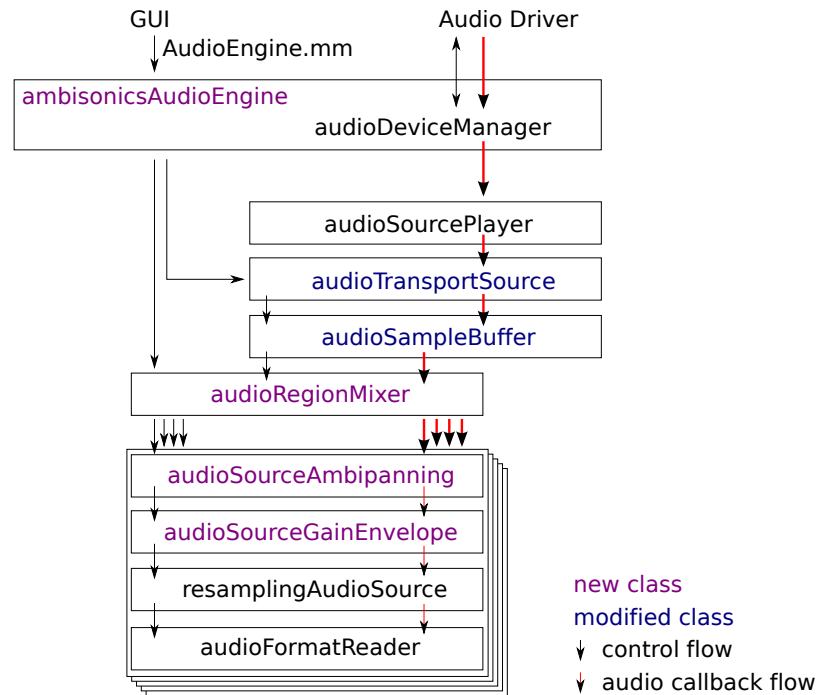


Figure 3.1: Audio Engine Overview

3.3.1 AmbisonicsAudioEngine

All actions of the audio engine are initiated by methods of this class. It's the hub of the audio engine. It also contains an instance of the `AudioDeviceManager` class, which communicates with the audio driver and provides a unified interface for it. The `AudioDeviceManager` can also be used for setting up the properties of the audio interface and for getting its status information.

3.3.2 AudioSourcePlayer

This is a wrapper class that converts audio callbacks as in Listing 3.2 to audio callbacks as in Listing 3.3.

3.3.3 AudioTransportSource

Provides a continuous stream of samples. If it is in the “stop state” it provides samples with value zero to the caller of its callback method and doesn’t call any callback methods itself. In “play state” it keeps track of the position of the virtual playhead and crafts its callbacks accordingly.

This class had to be modified because it was only capable of handling buffers with equal or less than two channels.

3.3.4 AudioSampleBuffer

As the name gives away, this is the place where the buffering of the audio stream takes place. Buffering is done in a separate thread with lower priority than the callbacks of the previously mentioned classes. Buffering is needed because the response time of the hard drive to a read request for a block of an audio file is in general bigger than the time interval determined by the frequency of the occurrence of the audio callbacks. The buffering class is at this position (and not closer to the `AudioFormatReader`) to avoid the need of having a buffering thread for every audio region present in the arranger view.

This class had to be modified because it was only capable of handling buffers with equal or less than two channels.

3.3.5 AudioRegionMixer

Contains an array of `AudioRegions`. An `AudioRegion` is a struct that basically consists of a pointer to an `AudioSourceAmbipanning` instance and the information about the position, length and the offset of the actual start sample of the audio file and the start sample of the audio region in the arranger view. Listing 3.4 shows the definition of this struct. For every audio region in the arranger view there is an entry in this array. At an arbitrary position of the virtual playhead, this class figures out which audio regions have a non-zero intersection with the current callback audioblock and sums there signals.

Listing 3.4: AudioRegion

```

struct JUCE_API AudioRegion
{
    /** The ID of this region. */
    int regionID;

    /** The position on the timeline, where this region starts.
        Specified in samples. */
    int startPosition;

    /** The position on the timeline, where this region ends.
        Actually this specifies the first sample after the end
        of the region. Specified in samples. */
    int endPosition;

    /** This is equal to startPosition - startSampleInTheAudioFile */
    int startPositionOfAudioFileInTimeline;
    // must be smaller or equal to the startPosition !!!

    AudioSourceAmbipanning* audioSourceAmbipanning;
};

```

3.3.6 AudioSourceAmbipanning

The main competence of the Choreographer, the spacialisation of audio, is done here. The signal for a speaker is calculated according to multiplying the source signal with the factor

$$\left(\frac{2 + x_{\text{speaker}}x_{\text{source}} + y_{\text{speaker}}y_{\text{source}} + z_{\text{speaker}}z_{\text{source}}}{2} \right)^M \cdot g_{\text{distance}}, \quad (3.1)$$

where g_{distance} is a function of the distance of the virtual source to the center - to simulate the distance from the source to the listener. The calculation (3.1) is actually only done on vertices of the spacialisation envelope, values in between are linearly interpolated.

3.3.7 AudioSourceGainEnvelope

A user defined gain envelope is applied to the original sound file. In contrast to some commercial available digital audio workstations (DAW), this gain envelope is sample accurate.

3.3.8 `ResamplingAudioSource`

If the sample rate of the audio interface differs from the sample rate of the audio file, this class takes care of the required resampling.

3.3.9 `AudioFormatReader`

Last but not least, the `AudioFormatReader` reads the audio file from the hard drive.

Chapter 4

Future Work

4.1 Improvements

- Addition of a speaker configuration dialog window.
- Implementation of the loop functionality.
- Write a user manual.

4.2 Additional Features

- Enhanced distance simulation by lowpass filtering.
- Support for audio regions in the Ambisonics B-Format.
- Audio export of the speaker signals to a multi channel file.
- Audio export to a Ambisonics B-Format file.
- Realtime control of the position of a certain audio region, e.g. with a hardware controller.
- Headphone mode, where the soundfield of the chosen speaker setup is emulated. This would enable a composer to work with the Choreographer without the need of an expensive speaker setup at home.
- Configurable delay and gain for every speaker, to compensate if a speaker is not placed at its ideal position.

Appendix A

Terminology

arranger view

A window of the Choreographer, that enables the user to arrange the audio regions on the horizontal time axis.

audio engine

The part of the Choreographer that is responsible for the whole signal processing.

audio region

A section of an audiofile that can be freely placed in the timeline in the arrange view. A gain envelope as well as a spacialisation envelope can also be part of an audio region

Choreographer

A piece of software to place audio records as virtual sound sources into space. The Choreographer feeds an array of speakers which simulate the sound field of these virtual sources.

GUI

Everything of the Choreographer, that is not the audio engine. GUI originally stands for the graphical user interface, but here this term is used a little differently.

Bibliography

- [1] JUCE. <http://www.rawmaterialsoftware.com/juce.php>.
- [2] Martin Neukom. Ambisonics einführung (only in german).
- [3] Martin Neukom. Ambisonics equivalent panning aep.
- [4] Jerome Daniel, Rozenn Nicol, and Sebastien Moreau. Further investigations of high order ambisonics and wavefield synthesis for holophonic sound imaging. *Audio Engineering Society, Convention Paper*, 5788, March 2003.

