Wave Field Synthesis with Real-time Control

A Project Report submitted in partial satisfaction
of the requirements for the degree of

Master of Science

in

Media Arts & Technology

by

William Francis Wolcott IV

Committee in Charge:

Curtis Roads, Chair
JoAnn Kuchera-Morin
Stephen Travis Pope
Matthew Turk

September 2007
The Project Report of
William Francis Wolcott IV is approved:

________________________________________________________________________

JoAnn Kuchera-Morin

________________________________________________________________________

Stephen Travis Pope

________________________________________________________________________

Matthew Turk

________________________________________________________________________

Curtis Roads, Committee Chairperson

September 2007
Wave Field Synthesis with Real-time Control

Copyright © 2007

by

William Francis Wolcott IV
Acknowledgements

It is a pleasure to thank my committee, JoAnn Kuchera-Morin, Stephen Pope, Curtis Roads, and Matthew Turk for all their advice and support, and especially, Stephen Pope for introducing me to this field. Thanks goes to Alex Norman for his programming expertise. August Black saved me countless hours with the power of \LaTeX. The master debugger, Jorge Castellanos was a valued teacher and a collaborator throughout my time at UCSB. Finally, I could not have done this without the support of Leeza Pachepsky, my parents, and all the MAT professors, students and staff whose enthusiasm fosters an environment of exploration and education.
Abstract

Wave Field Synthesis with Real-time Control

William Francis Wolcott IV

Wave field synthesis is an acoustic spatialization technique for creating virtual environments. Taking advantage of the Huygens’ principle, wave fronts are simulated with a large array of speakers. Inside a defined listening space, the WFS speaker array reproduces incoming wave fronts emanating from an audio source at a virtual location. Current WFS implementations require off line computation which limits the real-time capabilities for spatialization. Further, no allowances for speaker configurations extending into the third dimension are given in traditional wave field synthesis.

A wave field synthesis system suitable for real-time applications and capable of placing sources at the time of rendering is presented. The rendering process is broken into logical components for a fast and extensible spatializer. A broad range of users and setup configurations are considered in the design. The result is a model-based wave field synthesis engine for real-time immersion applications.
Curtis Roads

Dissertation Committee Chair
# Contents

Acknowledgements ........................................ vii
Abstract .................................................. ix

1 Introduction ........................................... 1
  1.1 Motivation ........................................... 2
  1.2 Problem Statement ................................... 2
  1.3 Project Goals ......................................... 3
  1.4 Related Work ......................................... 4
  1.5 Definition of Terms ................................... 8
  1.6 Document Overview ................................... 9

2 WFS Concepts .......................................... 11
  2.1 Theoretical Basis ..................................... 13
  2.2 Practical Considerations .............................. 14
  2.3 Speaker Driving Function ............................. 16

3 Application Considerations ............................ 19
  3.1 Users .................................................. 20
  3.2 Hardware Requirements ............................... 24

4 Design ................................................... 27
  4.1 Spatializer and Interface ............................. 28
  4.2 WFS Renderer Design .................................. 28
  4.3 Speaker ................................................ 29
  4.4 Room ................................................... 29
  4.5 Engine ................................................ 30
  4.6 Speaker Layout ....................................... 32
4.7 WFSynth Interface .............................................. 32
4.8 Interface ............................................................ 33

5 Implementation ...................................................... 35
  5.1 Engine ............................................................. 35
  5.2 Room ............................................................... 38
  5.3 Speaker ........................................................... 40
  5.4 Jack interface .................................................. 43
  5.5 OSC messaging ................................................ 44
  5.6 Speaker Layout ................................................. 46

6 Conclusion .......................................................... 51
  6.1 Results ............................................................ 53
    6.1.1 CPU Usage ................................................ 54
    6.1.2 Scale and Reuse ......................................... 55
    6.1.3 WFS into the third dimension .......................... 56
  6.2 Future Work ................................................... 56

Bibliography ............................................................ 59

Appendices .......................................................... 61
  .1 Example Speaker Layout File ................................. 63
Chapter 1

Introduction

There is an idea, the basis of an internal structure, expanded and split into different shapes or groups of sound constantly changing in shape, direction, and speed, attracted and repulsed by various forces.

Edgard Varese

Music is the arithmetic of sounds as optics is the geometry of light.
Claude Debussy

As the quest for realism in virtual environments continues, the need for synthesized sound, indistinguishable from a real sound, takes root. Wave field synthesis (WFS) offers just that, by simulating the exact conditions of an imaginary sound. However, the elegance of simple mathematical concepts quickly deteriorate to computational and programmatic complexity, demanding deeper analysis for practical use.
1.1 Motivation

The most significant feature of wave field synthesis is the enlargement of the typical reproduction sweet spot to a large listening space. Other features make wave field synthesis a promising spatializer. For instance, the first reflections of room acoustics are easily applied to a sound for an arbitrary room shape. Sweet spot oriented audio mixed for traditional stereo or surround is not only backwards compatible with wave field synthesis, but the sweet spot can be enlarged to the entire listening space.

Many reason exist to justify WFS as a means of virtual audio localization, yet the technique is rarely used due to the numerous speaker requirement. The few existing implementations do not take advantage of the potential of moving sound objects for true immersion and are built around specific uses and platforms. Wave field synthesis is conceptually simple and practically complex; I intend to hide its complexity, leaving only a conceptual spatializer to accommodate many applications and virtual environments.

1.2 Problem Statement

This project focuses on two challenges of current wave field synthesis rendering design, real-time processing and user interaction. Computational power has
caught up to the demands of WFS, but current techniques [15] [5] apply limiting off-line calculations. Rendering wave field in real-time requires a different approach, one that is capable of handling the computational load without stripping features and usability.

For virtual environments, user interaction comes in two forms: application design and room construction. The spatializer must be flexible to physical constraints of assembling a listening space and the wide range of possible applications that could drive a wave field synthesis renderer.

1.3 Project Goals

The goal of this project is to design and develop a complete wave field synthesis spatializer for interactive and immersion applications. Satisfying the needs of interactive spatialization proves to be the greatest challenge of such an undertaking. First and most importantly, interaction must happen in real-time. As demonstrated with experiment by Wenzel, [20], latency between audio and other stimuli is perceived at 250ms or smaller. Any jitter between audio and visuals or user interface is unacceptable for immersion environments. The WFS spatializer must hold its system latency to a perceptually invisible minimum.
Chapter 1. Introduction

For ideal interaction with spatial audio, there should be no limitations on how sources are spatialized. Sources should be free to move to any arbitrary point in free space at any time. Other WFS implementations [5] and [15] generate precomputed specific source locations prior to runtime, where each source must exist during spatialization. An interactive application requires the spatializer to spatialize audio in according to real-time input.

Immersion applications come in many forms, many of which cannot be anticipated. However, this project aims to serve as a spatializer for a range of users and environments. The spatializer will accommodate many user applications and attempt to be free of platform specificity that would limit its use. Users should be able to take advantage of WFS with any type of application or method. Likewise, immersed environments vary dramatically. Linear arrays of speakers, circular arrays as in CARROUSO [15], or the spherical environment of the AlloSphere [14] all demand different wave field synthesis specifications. The spatializer should be independent of speaker configurations and irregularities.

1.4 Related Work

The outlining concepts of wave field synthesis have existed since Berkout’s work [6] in 1988, but the computational complexity and expense has made WFS
impractical until recently. For that reason, realization of a wave field synthesis
spatializer is limited to a handful of instances. This chapter will provide an
overview of spatializers designed for wave field synthesis.

WONDER

Notably, Marije Baalman’s WONDER [5] is open source software for compo-
sition and playback of spatialized source data with WFS. WONDER records the
object’s motion with points along its path. For each point, a filter is calculated
prior to the WFS renderer’s runtime with the filter library BruteFIR [19]. When
an object moves between points, the resulting audio is panned to simulate the
motion.

There are several reason why WONDER’s method of rendering WFS is not
suited for many interactive audio spatialization environments. First, new source
positions cannot be defined in real-time. Once the filters are established, wave field
synthesis cannot be rendered at any other location. Next, moving, or panning,
a source between points does not render the source at some point in between.
Instead, accurate spatialization of this method will appear to be two independent
sources. In the case of a source constantly moving, at no time will there be correct
WFS rendering. Finally, the WONDER panning method eliminates the Doppler
Chapter 1. Introduction

shift inherent in wave field synthesis. Doppler shift may not be wanted in every case, but it does provide an extra localization cue to the listeners.

WONDER does provide a simple interface for composing and recording sound object motion for later playback. However, BruteFIR exclusively runs on Linux and thus WONDER is restricted to Linux platforms only. WONDER runs on JACK as an audio input method and uses OSC to receive control data for virtual sources.

CARROUSO

The European Union CARROUSO Project [18] is a system for creating a realistic sound field in a given listening space using wave field synthesis. Spatialization is achieved with virtual point sources both outside the listening space and inside (known as focused sources). Additionally, plane waves are available as another type of audio source. CARROUSO uses the MPEG-4 standard as a storage format for source audio and positional data. Beyond the WFS software, the CARROUSO Project includes a physical speaker arrangement. The speaker configuration has gone through many changes since the project began. Initially, a tight 48 speaker circle for 4 to 8 listeners demonstrated the virtual sound. Later, linear speaker arrays and multi-actuator panels (MAPs) are used. At the time of this publication, CARROUSO is not open for public use.
Similar to WONDER, CARROUSO uses BruteFIR to compute filters and lacks the ability to freely move sources in real-time. In addition to being a closed source project, the MPEG-4 standard is proprietary and requires permission for general use. However, MPEG-4 is well suited to the audio source model needed in most spatialization technique and similar alternatives are not apparent.

**IOSONO**

IOSONO [1] is a commercially sold WFS system designed for accurate playback of spatialized audio. Speaker arrays, rendering software, and driving sound equipment is sold as a full WFS package. As in all wave field synthesis implementations, IOSONO is capable of enlarging the sweet spot of audio mixed for a much smaller number of speakers.

While none of these existing systems satisfies the requirements of those laid out in the Goals chapter, they each have traits of the ideal WFS spatializer. CARROUSO establishes a computationally reasonable and diverse set of source types for a virtual space. WONDER, while predetermining the possible positions, does give real-time control of sources within this set of points through the messaging system OSC. Both WONDER and CARROUSO serve as building blocks to a better system for interactive audio immersion.
1.5 Definition of Terms

Several terms will be repeated throughout this document and should be clearly defined.

WFSynth

"WFSynth" is the basis for this project, and more specifically, a collection of processes rendering wave field synthesis. These processes are the Engine, Room and Speaker. The Engine is responsible for the initial computation. The Room is in reference to a speaker layout or speaker configuration. The Room processes data at the level of all speakers in a WFS array. Finally, the Speaker acts as a processing unit directly in parallel to a physical speaker.

speaker

Aside from an actual speaker used in a wave field synthesis array, the term, Speaker, is used in this paper to refer to a specific set of processes in the rendering chain. The WFSynth contains a class named Speaker which handles data processing correlating to each actual speaker. When referring to processing component of a speaker, the term Speaker will be capitalized.
Source

A source, or sound object, as it relates to model-based WFS is a virtual audio emitter placed in space with a specified radiation pattern. In this project, audio sources are monophonic streams of data accompanied by position information.

1.6 Document Overview

The mathematical constructs of wave field synthesis are outlined along with many of the practical constraints in Chapter 2. Following, an analysis of use cases and speaker layouts for the design of an interface is given. Chapter 4 provides the design leading up to Chapter 5, the implementation of the wave field synthesis renderer. Finally, results and conclusions of its performance and usability are stated.
Chapter 2
WFS Concepts

Wave field synthesis is a psychoacoustic spatial audio technique for simulating sound sources at a location in space. Invented by A.J. Berkout [6], it is based on the Huygens principle. Huygens principle (see figure 2.1, [11]) states each point along a wave front can be viewed as emanating a new wave front. Using this concept, it is shown that speakers can be placed over a surface to reproduce any wave traveling across the surface.

From the outlying wave field synthesis concept, two branches diverge, model-based and data-based WFS. Data-based wave field synthesis uses a microphone array to create a wave field recording, also known as wave field analysis. Impulses are measured to found that represent room acoustics, radiation patterns, and radiation direction. The resulting multi-channel recording can be transformed or modified according to the WFS principles to spatialize the source for a similarly large speaker array.
Chapter 2. WFS Concepts

Figure 2.1: Huygens’ Principle: Plane wave refraction on a perforated wall.

Model-based wave field synthesis simulates virtually spatialized sound sources by calculating a transform for a monophonic audio source with its given parameters in space. Traditionally, monopole points and plane waves are modeled due to their simplicity. With an audio source positional information and correct accounting of sources, a signal can be determined at each speaker in the WFS array, see figure 2.2 [13]. This project focuses on model-based wave field synthesis as a method of producing a virtual environment.
Theoretical Basis – Section 2.1

Figure 2.2: Two virtual audio point sources spatialized with a model-based WFS rendering.

2.1 Theoretical Basis

The Kirchhoff-Helmholtz integral is the basis for modeling a volume enclosed by speakers. The integral states if the pressure and velocity of an incoming wave is known for the surface of a source-free volume, the pressure is determinate for any point inside the volume.

\[ P(w, z) = \int \int_{dA} G(w, z|z') \frac{\partial}{\partial n} P(w, z') - P(w, z') \frac{\partial}{\partial n} G(w, z'|z') dz' \]  

(2.1)
Chapter 2. WFS Concepts

\( P(w, z) \) is the pressure inside the listening space in the Fourier domain at a point \( z \), \( G \) is Green’s function, and \( P(w, z') \) is the reproduction pressure needed at the surface of the listening space. An added feature of the integral is no conditions are made on the shape of the bounding volume, as long as the boundary conditions are met. The Green’s function is used to satisfy boundary conditions,

\[
G(w, z|z') = \frac{1}{4\pi} \frac{e^{-j\frac{w}{c}|z - z'|}}{|z - z'|}
\]  

(2.2)

where \( w \) is the wave number, \( c \) is the speed of sound, and \( |z - z'| \) represents the vector between a point on the surface to a point inside the volume. The Kirchhoff-Helmholtz integral with Green’s function is the basis for wave field synthesis, but not sufficient for developing an actual system of spatial reproduction. Rabenstein, Spors and Steffen [15] describe three practical constraints that must be considered.

2.2 Practical Considerations

The Green’s function used as the volume’s boundary condition demands both monopole and dipole speakers for a speaker array. While traditional speakers can be reasonably assumed to be monopole emitters, dipole speakers add an additional layer of complexity and expense to a speaker layout. Rabenstein proposes a windowing function as a substitute for dipole speakers.
Creating a listening space for several listeners with an enclosed speaker array would require more computational power than currently available by traditional means. The principles of wave field synthesis are distilled to 2 dimensions foster a more feasible spatial audio system. Reducing the problem to a line or circle array of speakers in the plane of the listener’s ear can be a sufficient for immersed audio. For full 3-D audio with a 2-D WFS array, HRTF [10] could be used to simulate sources outside the ear plane. Pope [14] [4] proposes a 2-D wave field array with Vector Based Amplitude Panning in the third dimension.

According to the Kirchhoff-Helmholtz integral, the speaker array across the surface of the enclosed volume is a continuous distribution. The discretization of the continuous surface to discrete speakers results in spatial aliasing. Similar to the more well known temporal aliasing, spatial aliasing causes incorrect spatialization above the aliasing frequency. The aliasing frequency is determined by the spacing between speakers in a linear array.

\[ f_{\text{alias}} = \frac{c}{2\Delta \sin(\alpha_{\text{max}})} \]  

(2.3)

where \( c \) is the speed of sound, \( \Delta \) is the distance between speakers, and \( \alpha_{\text{max}} \) is the maximum angle of incidence for incoming waves. For example, a speaker separation of 10cm produces aliasing at 1700Hz. Fortunately, unlike temporal aliasing, spatial aliasing is less pronounced perceptually.
2.3 Speaker Driving Function

The elimination of dipoles, discretization of loudspeakers, and reduction to two dimensions gives a speaker driving signal for both point and plane sources. Radiation patterns for a monopole is similar to the free-field Green’s function driven by the source signal in the Fourier domain,

\[ P_0(w, z|z') = F(w, \theta) e^{-j\frac{w}{c}|z-z'|} \tag{2.4} \]

A plane wave is seen as,

\[ P_0(w, z) = F(w, \theta) e^{-j\frac{w}{c}(x,n_\theta)} \tag{2.5} \]

with \( n_\theta \) the normal vector of the plane. The resulting driving signal [15] for a speaker at a speaker is

\[ D_\theta(w, x|x') = 2w(x', \theta)A(|x - x'|)K(w)e^{j\frac{w}{c}|x', n_\theta|}F(x, \theta) \tag{2.6} \]

\( w(x', \theta) \) is a window function as a result of eliminated dipole speakers and the normal dot product of the incoming wave. \( A(|x - x'|) \) is the amplitude attenuation due to distance and the reduction to 2 dimensions. \( K(w) \) is a square root of the wave number spectral shaping also due to the dimension reduction. \( e^{j\frac{w}{c}|x', n_\theta|} \)
applies the appropriate delay to the incoming wave. Finally, $F(x, \theta)$ is the signal emitted from the source.
Chapter 3

Application Considerations

A wave field synthesis renderer acts as the interface to user control and hardware output. While wave field synthesis is a mathematically defined method, both the user input and system setup output can be handled in a variety of ways. Software design considerations must account for both user and hardware sides of the WFS renderer for best usability.

While most software systems must consider their user as the driving force behind design, a wave field synthesis renderer is unique in that its system requirements also bring large demands. The two main restrictions of a WFS system, computational complexity and large numbers of loudspeakers, force hardware and software configurations to be different for each particular system. A critical component of the design of a wave field synthesis renderer is the flexibility to handle a variety of hardware and speaker setups.
Chapter 3. Application Considerations

Likewise, the spatializer must be simple for the user while exposing all the benefits of wave field synthesis. The primary goal is to allow a broad range of potential users to control the WFS renderer at a level based on their needs and abilities. This chapter surveys the different system requirements and use cases for a wave field synthesis renderer.

3.1 Users

In this section, several use cases are presented that motivate the design of this wave field synthesis renderer. The WFSynth is intended for real-time model-based wave field synthesis with the broadest range of users as feasible. The WFSynth is meant to encapsulate users from those with limited spatialization knowledge to audio researchers. There are several groups for which a WFSynth is aimed:

- Researchers of Multi-channel systems
- Musicians interested in spatialization
- Spatial audio tool software designers
- Immersion artists in need of audio spatialization
- Researchers simulating multi-channels systems or room acoustics
- Wave field synthesis researchers
Researchers of Multi-channel Systems

An audio system designer is building a multi-channel system. She wishes to try many spatialization techniques to see which works best for the particular system. The system designer wants an application she can run and quickly hear a simple test of WFS in her multi-channel setup. She is assumed to have little or no programming or spatialization knowledge and requires a simple interface for testing.

Musicians Composing for Spatialization

A musician composes a work in a multi-track application such as DigiDesign’s Pro Tools or Audacity. Rather than mixing the score down to a defined number of channels, she leaves the score in its multi-track elements. The musician intends to position each track in space as a part of a real time performance.

The movement of each source could be previously recorded or performed in real-time. Two requirements must be fulfilled in order to make this scenario possible, previously recorded audio and real-time position control. When moving an audio source, the WFSynth must not have regions of inconsistency or error. Speaker configuration and the principles of wave field synthesis will always dictate where and how sound is rendered, but a well-formed WFS renderer will offer seamless transitions when a sound object reaches boundaries of the system.
Chapter 3. Application Considerations

Spatial Audio Tool Software Designers

A software programmer wishes to integrate wave field synthesis into a rich set of spatialization tools. These tools might include: a WFS composition user interface, an off-line rendering tool for later playback, a spatialized file format converter through WFS.

In these cases, the audio tool designer needs to be able to integrate the WF-Synth in code. The Engine will give direct access to input audio and control values and return rendered output based on the user supplied speaker configuration. The API should cross as many platforms as reasonable and be available on a standard and fast language. Additionally, the WFSynth would be more usable to a software designer if the API and its dependent libraries are open-source and come with flexible licenses.

Immersion Artists in Need of Audio Spatialization

An artist or designer is creating an interactive audio and visual exhibit and chooses wave field synthesis as the method of immersion for the participants. Audio is generated in real-time along with the spatial positioning of these audio sources. Audio and control messages may be generated by the installation or supplied interactively by participants.
For real-time interaction, a critical component is low latency relative to user response. If audio doesn’t respond to the user at the same rate as visual and haptic feedback, the user interface can be confusing and disorienting.

Furthermore, the WFS renderer should be open to control on different levels. For cases where WFS works in parallel with video and human interaction, many times, multiple machines are employed. The WFSynth should be available for audio and/or control signals to be sent over a network.

**Researchers Simulating Multi-channels Systems or Room Acoustics**

Wave field synthesis has the potential to recreate many listening scenarios. Room reverberation and multi-channel simulations are two of the areas of research resulting from WFS. As an example, an audio engineer is mixing audio for conventional home sound systems. A WFS renderer could quickly reproduce a variety of multi-channel setups such as stereo, Dolby 5.1, 7.1, etc.

Wave field synthesis offers room acoustic simulations in the same way. A software designer works with an architect to integrate the WFSynth with a CAD drawing. The architect can modify the building drawing and hear the changing acoustics in real-time.
Wave Field Synthesis Researchers

Finally, a researcher testing new techniques for wave field synthesis wants a previously built framework on which to add her new algorithm. A well organized open-source WFSynth could be used as a building block, test-bed, or comparison for new WFS research.

3.2 Hardware Requirements

Wave field synthesis lends itself to a complex hardware and speaker system. Existing literature [16] shows what aspects of a WFS will give the ideal audio reproduction. Perhaps the most obvious prerequisite is a large number of closely-spaced speakers. While certain speaker configurations give better spatialization, the WFSynth should not discern or limit any speaker setup. A two channel system, while insufficient for accurate WFS, still provides intelligible spatialization when render through WFS. Very few multi-channel setups will be able to recreate the ideal tightly spaced circle array needed for WFS. However, the WFSynth can accommodate for speaker and hardware irregularities. A few special system configurations are discussed:
Line and Circle Speaker Arrays

Audio sources positioned outside the speaker array are handled no matter the geometry. However, for source holography inside or in front of a speaker array, the WFSynth must handle rendering differently. The WFSynth should be designed with both line and circle geometries in mind.

Speaker Configurations into the Third Dimension

While three dimensional wave field synthesis is a topic for further research, the WFSynth can accommodate speakers set outside the ear plane. This can be useful for two reasons. First, physical speaker placement limitations can be compensated for with negligible reproduction error. Second, a system designer may choose to place speakers in three dimensions using wave field synthesis for exact reproduction in the 2-D plane and WFS-esque spatialization in 3-D. The rendering in the third dimension would be mathematically inaccurate, but a WFSynth should continue to render WFS to obtain spatialization at different elevations.

 Speakers Positioned Outside the Ideal Line or Circle

Similar to the last case, it may be possible to position speakers within the ear plane but in an exact line or circle. The WFSynth should correct for slight differences in alignment.
Irregular Spacing Between Speakers

As in Rabenstein [15], there is no requirement as to the exact spacing between speakers in an array. However, irregular spacing requires amplitude adjustments for accurate audio reproduction. The WFSynth offers compensation for irregular spacing.

Sound Card Specifics

Many sound cards have several kinds of outputs. A particular system may not want to use specific outputs or want to reorder outputs in software rather than reorder physical cable connections.

Operating System/Platform

To encompass as many user and systems as possible, a WFSynth needs to run on the most common operating systems, programming languages, and protocols.

Designing a WFSynth for potential users with a broad range of computer acumen opens the unfortunate possibility of alienating users at some level. To counter this, the WFSynth is model with intuitive layers, each hiding a level of complexity, that can be peeled back by users of different need and ability. By offering both a user interface and programming interface, the WFSynth is open to any user interested in spatialization.
Chapter 4

Design

Based on the user and hardware requirements outlined in the previous section, a model for the WFSynth can be developed. The primary goal of this spatializer is to make available a two-level user interface to model-based wave field synthesis such that hardware and speakers are decoupled from user interaction. The multi-tiered user interface is a combination of a low-level API and JACK plug-in for higher level pluggable spatialization. At any level, the WFSynth only requires audio object information on the user front-end. In this way, the user will need no knowledge of wave field synthesis to effectively spatialize audio. Moreover, this model hides the hardware and speaker details and operations so that a WFS system can be transferred without reconstructing the user’s configuration.
4.1 Spatializer and Interface

The WFSynth is designed to be the bare minimum required interface for wave field synthesis rendering. The Engine application is strictly the procedural DSP chain which converts source audio and control data to speaker output based on a speaker layout.

With the most basic wave field synthesis rendering framework in place, a facade can be wrapped around the WFSynth to provide user interaction and hardware control. The separation of renderer and interface is not only intuitive, but allows a removable layer in the wave field process depending on the user’s needs. While the wrapper around the Engine is a critical component, this project focuses on the design a wave field synthesis renderer.

4.2 WFS Renderer Design

There are three concepts, stemming from the spatial audio work of Castellanos [7], on which the WFSynth is modeled, the Engine, Room, and Speaker. Each of these three components is its own class (see figure 4.1), they make up the procedural and logical path for start-to-finish wave field synthesis rendering.
4.3 Speaker

Within the context of the WFS renderer, the Speaker is an audio processing class paralleling a physical speaker. Each instance of a Speaker contains information of the actual speaker’s position and direction. The Speaker is the final stage of processing chain, doing audio computations specific to each speaker. Each Speaker saves blocks of audio such that when all processing completes, its data can be sent to the sound card and physical speaker.

4.4 Room

The Room class is a container for each instance of a Speaker. Its contribution to the renderer is small but significant; the Room bridges audio sources and
Chapter 4. Design

Speakers. In this way, a speaker layout and configuration is contained concisely within one Room, simplifying interaction with audio sources. The Speaker needs not have knowledge of the audio source and vice-versa.

As a secondary task, the Room class has the broadest overview of the speaker layout and uses it to determine special case phenomenon. More specifically, the Room calculates whether a source is inside or outside the listening space. This calculation may be different for unique speaker configurations. The Room class is sub-classed in order to handle circular and linear speaker arrays. Additional sub-classing of the Room class is necessary to accommodate spherical or planar speaker arrays.

4.5 Engine

Finally, as the first stage of the WFS renderer, the Engine accepts audio source data and control messages. Any processing or logic applied specifically to audio sources takes place within the Engine. Spectral filtering, error checking, amplitude adjustments are made in this stage. The Engine is isolated from the other two components in the renderer, only passing its modified audio source data to the Room class when finished.
The Engine, Room and Speaker class design is important for eliminating redundant computation and outlining an extensible method for large-scale WFS rendering. In the case of Rabenstien [15], the DSP chain is said to be a $m \times n$ matrix multiplication of audio sources ($m$) and speaker ($n$). WFS implementations such as WONDER by Balmann [5] follow Rabenstein’s method by constructing a filter for each source and speaker combination leading to, again, $m \times n$ filters. Much of the filtering is redundant in a real-time situation and is reduced by the Engine and Room model outlined.

Secondly, the Engine, Room, and Speaker model creates a theoretical framework where components of the renderer could be placed on different processors. Because the audio sources are encapsulated inside an Engine and, likewise, the Speakers are contained in a Room, multiple instances of either class could exist as a part of a larger WFS renderer. For example, a collection of computers are limited by each CPU being able to handle 32 channel output. With networked and synced output, each computer could contain a Room of 32 Speakers where a single Engine supplies audio source data. Similarly, for large numbers of audio sources, many unique Engines could be distributed passing processed data onto a single room. Continuing the example, a combination of both could exist with multiple Engines networked to multiple Rooms.
4.6 Speaker Layout

A list of speakers is necessary before any wave field synthesis rendering can take place. Speaker position and normals are the basis for the creation of each instance of a Speaker. Additionally, a group of speakers can be attributed to a particular Room, if multiple Rooms are defined. When an instance of an Engine is created, a speaker layout file is read and the Room and Speakers are created in turn.

4.7 WFSynth Interface

All WFS rendering is handled by the Engine class. In order to create a spatializer which is as simple and compact as possible, programming access to the WFSynth only comes in two forms, initialization and callback. The speaker layout is given to the Engine at initialization and the Room and Speakers are constructed as described above. The callback method is where audio and control input data is given and output buffers are returned.
4.8 Interface

The rendering engine modeled here is designed around being concise, not for complete usability. An interface wrapper around the rendering engine satisfies many of the use cases for wave field synthesis. Several concerns need to be addressed in designing an effective interface. First, raw audio data as input, from a user and operating system standpoint, dramatically complicates a WFS system. Second, audio and control data is likely to be asynchronous. Finally, similar to the renderer’s input, the output has no connection to the physical speakers. These three issues are discussed in designing the interface.

At this point, a key assumption has to be made regarding the WFSynth’s stage in the audio chain. As in most spatializers, wave field synthesis is the last step audio passes through before being played back by the speakers. Since WFS theoretically gives exact sound field reproduction, there should be no reason to alter the spatialized audio after the rendering process. Of course, situations exist where this assumption is shown to be false. For example, a final process to correct for speakers spectral dynamics could be inserted after spatialization. Nonetheless, this type of example does not affect the task of the WFS spatializer and the original assumption can be held for the design of an interface.
Chapter 4. Design

Declaring the spatializer as the final processor before playback allows the audio source model for spatialization to play a powerful role in the WFSynth interface. The WFS interface can be a wired connection to the sound card and speakers so the user only need devise a plan for connecting sources to the spatializer. An ideal scenario is for an audio stream from any source (application, OS, or sound card input) to be routed as and spatial source for the WFS renderer. This can be accomplished through the open-source software, JACK [2]. In addition to handling audio streams as spatializer input sources, JACK is also a framework for connecting and reconfiguring speakers connected to the WFS spatializer.

Lastly, the interface provides a more robust messaging system than the WF-Synth. It cannot be assumed that positional control messages will be sent at an equivalent rate as with audio blocks. Handling this asynchronous data presents two cases, control messages sent faster and slower than block rate. Since the WF-Synth handles data in blocks, only one control message can be given to correspond with one audio source. Therefore, the interface should only pass the most current position update to the spatializer. For the case of messaging slower than block rate, the interface should preserve source positions as stationary.
Chapter 5

Implementation

The outlining concept of wave field synthesis appears elegant and straightforward, but implementing its principles in software exposes a number of compromises for a usable real-time system. The resulting WFSynth follows the specification laid out in the Design section and are described in greater detail here and shown in figure 5.1. The most important component, the wave field synthesis rendering spread between the Engine, Room, and Speaker class, is described. Additionally, the user interface to the project, the JACK interface, control messaging, and speaker layout, are outlined.

5.1 Engine

The Engine is the start of all WFS rendering and provides the only communication from the spatializer to an outside interface. An instance of a WFS spatializer is created in this way:

```
Chapter 5. Implementation

Engine myWFS(/path/to/configFile);

The text file configFile is read and the WFSynth is built. The resulting speaker layout and user options are used to created the correct type of Room and an instance of the Speaker class for each described in the configuration file. Options and parameters such as room shape, audio rate and latency options propagate to the Room and Speakers.

Once the Engine is configured, the callback method can be used for each buffer to be spatialized. An example rendering of one buffer of audio might look like this:

```java
myWFS.processBuffer(sourceData, outputData, x, y, z, type);
```

'SourceData' contains the raw audio data of sources. 'OutputData' is the buffer to be filled with rendered data. The positional 'x', 'y', 'z' are the locations corresponding to the audio given in 'sourceData'. Finally, the 'type' variable tells the WFSynth the shape of the audio source, either a point source or plane wave. Each argument of the processBuffer method is given as a vector. With the exception of outputData, each vector is the size of the number of sources to be rendered. This number need not be consistent; only the number of sources given are processed. The vector 'outputData' is the constant length of the number of output channels defined by the speaker layout.
Other than the intermediary to the outside, the Engine class takes on several rendering responsibilities. Two mechanisms check the source data for potential silence. First, input audio The is zero checked. Second, source position is measured against the listener position for a distance. If the distance is inside the range of being heard, the WFSynth continues. This not only acts as a safety check, but also adds a simple method of turning off a source, and the subsequent rendering, by placing it a substantial distance away.

The Engine filters each source’s audio according to the driving signal2.6. The square root filter is defined during Engine initialization, leaving only the discrete Fourier transform to be applied to the audio data. The overlap-add method is used for filtering and audio is transformed back into time domain. Each of the filtered overlap segments are sent to the Room to continue the DSP chain.

Using cached source position data, the Engine also checks if the source has moved since the last buffer. The Room has two calls for processing the next buffer, one for a static source and one for movement. If a source is motionless, the Room and Speakers can omit much calculation by using its own cached data.

For objects moving between buffers, the Engine can improve the positional accuracy of the sound. Due to the overlaps, the audio block rate is now divided by the number of overlaps defined. This allows for finer gradation of position.
Cached source positions are interpolated with the current position data to create a smoother movement transition.

## 5.2 Room

The Room’s task is to receive buffers of audio and distribute them to each Speaker. For the case of stationary sound objects, this is simple as passing the audio to the Speakers to use cached values for the final filtering stage. Moving objects require an algorithm for determining how to correctly handle the given source. The problem to overcome is how to determine whether the source is inside the listening space or outside. The Speakers must behave differently based on this condition. It is neither simple nor efficient to evaluate this at the Speaker.

The Room class, however, uses its broader perspective of the speaker configuration to make this determination. Upon initialization, the Room develops a map of the listening space which it uses during rendering to evaluate a source position. The shape of the speaker configuration plays a role in the formation of this map, and, thus, the subclassing of the Room into a LineArray or CircleArray.

For a LineArray of speakers, each speaker position is calculated in a least-squared interpolation of the best fitting line. This is used by the room as the boundary on which a source will be considered an ’inner source’. For irregular
shaped line arrays of speakers, a source is still treated as an 'outer source' within the Speaker if it lies inside the Room’s line boundary but is outside a specific speaker. However, this special case is handled in the Speaker class.

Finally, the Room tests the source position for being outside the entire listener space. Any source positioned behind the ideal listener cannot be spatialized using the WFS method and is muted by the WFSynth.

A CircleArray of speakers is treated in an equivalent manner in the Room as the LineArray. A circle is mapped based on the most distant speaker from the ideal listener. Sources inside this circle are evaluated as inner sources. The only exception is the Room enforces no forbidden source placements in the way the LineArray class does. There is the assumption that the circle is fully enclosed and therefore a source is always inside or outside the speaker array. However, this may not be true for every setup. For example a half circle array will have a similar requirements to a line array, where half the possible positional area is forbidden for sound sources because they cannot be correctly spatialized. While this issue is not confronted directly by the CircleArray, the inner source area for a CircleArray is limited to the area in which the speakers can correctly spatialize a source. Outside this area, the wave field synthesis principles handle the rendering, or lack thereof, of a source with a half circle array.
Circle and line arrays are descriptors only of the general form of the WFS setup. The entire rationale behind mapping the listening space is to correctly render audio for irregular speaker layouts. The physical constraints that require speakers to be placed outside their ideal position is acceptable for WFS. However, the LineArray and CircleArray are implemented for cases where there are speakers that must be placed outside of a straight line or there is an enclosed speaker array that makes a more irregular shape such as an ellipse or arc.

5.3 Speaker

An instance of a Speaker class is a generic object in the WFSynth. It has the same behavior for any shape speaker layout and does not depend on or have any knowledge of its owner, the Room and Engine class. Upon initialization, the Speaker only needs to store its location information and build a circular buffer in which the final rendered audio will be stored.

For each buffer of audio, the Speaker class is responsible for the remaining components of the filter, the normal vector attenuation, distance attenuation, distance delay and speaker specific gains. The attenuation is applied to the source audio and the delay is realized by the point in which the audio is mixed into the circular buffer. Since the circular buffer is predefined, the delay and thus the
distance and object can be from the speaker is also predefined. In the case where
the sound object is placed beyond the limits of the circular buffer, the delay is
shortened to the maximum possible in the buffer. Implementing the WFS delay
cues in this way does present inaccuracies at long distances. However, in practical
cases, the circular buffer can be made sufficiently long enough to accommodate
an 'out of earshot' length delay with negligible additional memory requirements.

Applying the computed delay to the output speaker signal places the source
at the correct location only at one point in time per audio buffer. If a source is
moving relative to the speaker, a jitter will occur as the source position is updated
each buffer. The result is a subtle tearing for sound objects moving significantly
fast.

The Speaker class reduces this effect by assuming a constant velocity between
the current and previous position. Comparing cached position data against cur-
rent, the source’s velocity is determined. The velocity vector’s dot product with
the normalized relative position to the speaker gives $\Delta f$, the change in rate due
to the Doppler Effect,

$$\Delta f = \frac{fv}{c} \quad (5.1)$$

where $f$ is the normal rate of 1, $v$ is the speed relative to the speaker, and $c$ is the speed of sound. Upon writing the audio buffer into the circular buffer,
the Doppler shifted audio rate is used to correct for the source’s movement. This process is equivalent to recalculating the correct delay for each sample along an assumed linear path of the source.

The Speaker’s filtering algorithm is a bit different for plane wave audio sources. A plane wave can be thought of as a point source at an infinite distance emitting a signal not dampened by the infinite distance. Another way to view this type of source is as an infinitely extended flat plane of sound that will have the same loudness at any listening point. Neither of these explanations are feasible to implement directly into the software.

A method is devised to handle plane waves in a more rational way. The simplest implementation would be to have the plane wave have a user assignable location by which WFS is rendered. However, a plane wave has an additional property, direction of travel. For this concept to work, two accompanying data points, a location anywhere along the plane and normal vector, are needed rather than the one of point sources. This is unacceptable in that it would dramatically complicate the user interface and reduce usability.

One solution to this problem is to remove the concept of a position for plane waves, leaving only a normal vector. The plane wave is assumed to be at a distance from the ideal listener just behind the array of speakers. Now, each Speaker can determine its delay to correctly render the plane wave. There is the downside
that no delay can be placed on the plane wave. However, with the exception of a pure delay, the WFS rendered audio is identical no matter where is emanates from, unlike point sources. The feature of a delayed plane wave is sacrificed for ease of use.

With the Engine, Room and Speaker in place, the software implementation is a complete WFS renderer. All that remains is the user interface to encapsulate the spatializer.

5.4 Jack interface

JACK, short for Jack Audio Connection Kit, is an open source audio layer that sits between the sound driver and application level. Where applications normally pipe audio directly to the audio driver, JACK is able to intercept and reroute audio to and from either application or driver using in-ports and out-ports. In-ports could be physical speaker channels or a recording application’s input. Out-ports come in the form of line-ins from the sound card or the output of a software synthesizer. Where most applications connect directly to the in and out ports of the audio driver, JACK has the ability to connect any out port to an other in port.
Chapter 5. Implementation

JACK allows the WFSynth to become the last stage of audio processors before audio is directed to the array of speakers. The JACK wrapper connects directly to the speakers in the order specified by the speaker layout file given by the user. The vector of output audio buffers which is filled by the WFSynth is connected through the JACK wrapper to the sound card output.

Likewise, the JACK wrapper for WFS sets up a user specified number of inputs which act as ports for audio sources. Any application output or sound card input can be connected to the WFSynth as an audio source. Similar to the speaker output, each source’s buffer is organized into a vector for the WFSynth to render for wave field synthesis. The JACK wrapper for the WFSynth uses Alex Norman’s C++ wrapper [12] for the JACK API.

5.5 OSC messaging

The JACK audio source inputs are incomplete for spatialization without the accompanying positional and shape data. A message system, as described in the Design chapter, is implemented to complete the data required for the WFSynth. Open Sound Control (OSC) [3] is an open and extensively used protocol for sending network messages. OSC offers the added benefit of allowing the input audio generation and control to take place from any remote networked location.
As the last function of the JACK wrapper, OSC messages are read in by JACK’s lock-free ring buffer to assure messages are received in a thread-safe manner. All messages are stored while the WFSynth renders, and are recovered between buffers. Only the most recent messages are used. If more than one message describing an audio source is received, older messages are discarded. Similarly, if no OSC data is read regarding a source, the position data from the previous buffer is used for the source.

OSC messages must abide a specific format to be received correctly. Messages begin with the ‘/wfs’ header, followed by source number which is to be described, x, y and z position, source type and a user-defined gain associated with the audio. An example message might look as such:

```
/wfs 3 4.5 1.2 -7.6 1 0.77
```

This message moves the 3rd source to a position (4.5, 1.2, -7.6), declares it as a plane wave source, and drops the gain to 0.77. There is no message for turning off rendering for a particular source. However, dropping the gain to zero or sending a position sufficiently large does essentially that. The Engine class stops processing immediately if either of these conditions are met.
Chapter 5. Implementation

5.6 Speaker Layout

The final form of user input is supplying the WFSynth the position and normal vectors of the speaker layout. In addition, there are several other user defined values declared with the speakers. Both speaker layout and complimenting values are kept in a text file read upon WFSynth initialization. An example speaker layout file is given in appendix A.

Auxiliary variables are listed first in the layout file. The available user options are 'shape', 'weighting', 'delay', and 'sources'. As described in the Room section of this chapter, 'shape' declares how the speaker array is to be treated, either as a line or a circle.

'weighting' is in reference to a fixed gain for each speaker as described in 2.6. Wave field synthesis allows for irregularly spaced speakers provided the speaker is assigned a geometrically dependent gain. The speaker weighting option can be stated as equal, where a gain of 1.0 is given to each speaker. Otherwise, the user supplies a gain when describing each speaker.

Latency and reproduction accuracy are common trade-offs for a spatialization system. 'delay' is an option for which provides a choice between accurate inner source reproduction and reduced latency. The WFSynth requires an additional pre-delay to be able to place sources inside the listening space in front of speakers.
By choosing the lowest possible latency, inner sources will not be rendered by the WFS method. While all latency should be considered in designing a real-time system, this pre-delay is small relative to the latency introduced by the WFSynth block rate and sound card. It is recommended to use correct inner source rendering.

The 'sources' option tell the WFSynth the maximum allowed number of audio sources. This information also propagates to the JACK wrapper so that this number of input ports can be opened. If any of these listed options are omitted, default values take their place.

Once the user options are defined, each speaker is described. Each speaker lists its position and some point in space to which it points. While seemingly a bit strange, it is much simpler to determine a point in which the speaker is facing rather than its normal vector. This point will be the center of a circle for circular array or a point in front of the speaker line for linear arrays. The WFSynth uses the speakers position and facing point to calculate the normal vector. The final value for each speaker is its geometrically dependent gain, if that option has been set by 'weighting'.

When determining how the speakers are to be assigned in space, the ideal listener must be considered. The ideal listener is always lies on the origin, (0, 0, 0). The speaker array can be arbitrarily placed as long as the target listening
space contains the origin. Inner sources will not be calculated correctly if this condition is not met. For line arrays, inner sources are rendered up to the ideal listener and not beyond. For circle arrays, it is recommended the origin be the center of the circle.

There is a subtle distinction regarding the declared point in which the speaker is facing; this point should represent the direction in which the speaker should be facing and not the direction the speaker is actually facing. The direction it should be facing is in respect to the imaginary surface or line formed by the speaker array. Physical constraints may give rise to a situation where the speaker does not ideally face the listening space. However, in WFS, a speaker is assumed to be a perfect monopole. Thus, the direction it actually faces is immaterial. Moreover, this assumption generally holds for a speaker’s small angular offset.

There is one final option to accompany the list of speakers is the ‘skip’ value. For many sound devices, multiple types of outputs exist, some of which are not needed or used. The WFSynth creates a boolean list that contains which speakers are to be assigned to an output channel. This allows the WFSynth wrapper read this list and assign speakers to the correct outputs.
Figure 5.1: WFS Rendering Process Chart
Chapter 6

Conclusion

A wave field synthesis implementation has been presented that allows real-time redefinition of rendering filters and flexible interface for user applications and environment designers. The WFSynth renders a smooth transition for sources in the far-field approximation, near-field effect, and inside the listening space. Effective organization of filters minimizes the spatializer’s calculations allowing more sources and speakers in a stable system. Sources can be quickly connected and disconnected to the rendering engine and an open network protocol can provide source meta-data locally or from any location across a network. Both linear and circular speaker arrays are handled correctly despite positional irregularities.

Using the JACK server and OSC messages, virtual audio scenes can be rendered and modified in real-time. Delay is correctly calculated at a per sample level which constructs accurate Doppler shift for moving sources. Practical applications such as rendering early reflections of room acoustics (see figure 6.1) or
Chapter 6. Conclusion

the enlarging of a traditional recording’s sweet spot are built by placing point or plane sources in space. Rather than recreating real life acoustic scenes, unnatural scenes can be generated for a different effect. Figure 6.2 demonstrates plane wave sources from many directions giving an unfamiliar immersion to the listeners.

Figure 6.1: Virtual audio sources placed as room reflections behind a linear array of speakers.
6.1 Results

The WFSynth’s real-time filter calculation bridges the divide of latency requirements and spatializer robustness. The Fourier domain filtering adds anywhere from 6ms to 10ms of latency, depending on settings, to the operating system’s built in latency of around 30ms. The total latency fairs considerably better
than the perceived jitter effects of a 250ms delay. The JACK wrapper around the WFSynth adds no latency to the system.

6.1.1 CPU Usage

Equally important to the virtual audio system is the computational limits of wave field synthesis. The two factors that affect the CPU load are sources and speakers. Each additional source adds a DFT and IDFT for spectral filtering. Additional speakers add a significantly cheaper delay and gain per each source. On a MacPro (2 X 2.66GHz Xeon processor) used for testing, 20 sources rendered for 32 speakers consumes close to 80 percent of the available cycles, close to the capability of machine to provide consistent audio output. Experimentally, one additional source take around four times the CPU as one additional speaker. For 16 channel output with 20 input sources, the WFSynth stabilizes at close to 60 percent. The spectral shaping in the Engine class adds the greatest computational load to the rendering process, and developing techniques for condensing sources or reducing the number of DFT’s would be the greatest benefit to efficiency. Despite the computational level on which the WFSynth is running, the JACK client typically requires a negligible, less than one percent CPU usage.
6.1.2 Scale and Reuse

The resulting code base of the WFSynth contains about 2200 lines of C++ including the JACK and OSC wrapper. The project was designed with reuse in mind. However, many compromises were made to accomplish other goals such as efficiency, functionality, and user simplicity. For example, the WFSynth has poor generality because it was designed for the specific task of real-time wave field synthesis rendering. The WFSynth would not easily translate to topics similar to model-based WFS such as data-based WFS, Ambisonics, or other off-shoots of wave field synthesis.

The concept of safety plays an important role in usability. The WFSynth is designed to render seamless audio with very few conditions for spatialization. The result is a pluggable API resilient to unexpected behavior in many different uses and scenarios. An exception is in the case of CPU overloading. The WFSynth does not attempt to maintain stable rendering when heavily loaded. Compromising rendering quality for perceptual continuity is a relevant area of future work.

Finally, code simplicity determines the projects reuse. From a programming interface perspective, the WFSynth is constructed with only initialization and callback functions. A straight-forward speaker layout file defines the physical setup and STL-based vectors hold source data and positions. The minimal interface is designed for simplicity. The renderer becomes more complex within the specific
modules, Engine, Room and Speaker. Information passed between modules is small and particular modules can be broken apart to achieve specific tasks in the WFS DSP chain.

6.1.3 WFS into the third dimension

The ultimate goal of three dimensional wave field synthesis is not addressed in this project. However, third dimension spatialization can be accomplished with the WFSynth. If speakers are placed outside the ear plane, the same wave field principles are applied giving similar localization effects to the two dimensional case. Although, the WFSynth computes filters for the 2-D case, and thus, rendering will be incorrect for speakers outside of the ear plane. But this can be seen as an advantage; by using the same filters, rendered sources will sound seamless moving from the ear plane to an elevation. Such a situation is an ideal case for a geometric weighting of less densely spaced speakers over head a 2-D array.

6.2 Future Work

Topics branch wildly from the original tenets of wave field synthesis. Sound reinforcement, sound cancellation, microphone arrays, data-based rendering, all
are resulting areas of research. Several topics that would integrate well and be a useful enhancement of the WFSynth are discussed in this section.

3-D WFS

The technical hurdle of coordinating large arrays of speakers far outweigh the challenges of software rendering for such a system. However, large scale immersed audio systems as in [14] [4] demand the precise rendering of wave field synthesis together with overhead spatialization. 3-D WFS, hybrid WFS and ambisonics [9], or 2.5-D systems are an interesting area of future work.

Speaker Adjustments

The assumption that speakers are perfect dipoles hinders accurate reproduction of virtual audio. Spectral non-linearities, carotid radiation patterns, and varying spectrum at different angles can be accounted for with filtering techniques [17]. Additionally, rolling off gains at the ends of linear speaker arrays produce better results [8]. Better accounting of assumptions would result in more precise rendering of virtual sources.
Chapter 6. Conclusion

Source Shapes

The two types of sources available in the WFSynth, WONDER, and CAR-ROUSO are point and plane waves. While certainly the least complex, these two type are not the limit of the capabilities of wave field synthesis. An arbitrary shape can be defined with a numerical solution approximating its radiation pattern. As a first step, exploring simple forms, such as spherical sources, would be an encouraging start for more complex forms.

Distributed Processing

This project was designed as if it may be distributed across multiple processors in the future. The Engine and Room class pass minimal amounts of audio data between them such that each phase could be separated by a fast network connection. Large-scale systems would require research into efficient techniques for processing the sources separately from the speakers.
Bibliography


Appendices
.1 Example Speaker Layout File

// a comment
// these options should be first
// the speakers form a line (vaguely)
shape line
// each speaker assumed to be equally spaced,
// no gain adjustment needed
weighting equal
// maximum of 16 input sources
sources 16
// we want inner sources (sources in front of the speaker array)
// to be reproduced
delay inner
// here’s the list of speakers
speaker .23 1.0 0.0 .23 0.0 0.0
speaker .46 1.0 0.0 .46 0.0 0.0
speaker .69 1.0 0.0 .69 0.0 0.0
speaker .92 1.0 0.0 .92 0.0 0.0
speaker 1.15 1.0 0.0 1.15 0.0 0.0
speaker 1.38 1.0 0.0 1.38 0.0 0.0
speaker 1.61 1.0 0.0 1.61 0.0 0.0
speaker 1.84 1.0 0.0 1.84 0.0 0.0
// skip these next two channels.
// maybe they’re outs that we’re not using.
skip
skip
speaker -1.61 1.0 0.0 -1.61 0.0 0.0
speaker -1.38 1.0 0.0 -1.38 0.0 0.0
speaker -1.15 1.0 0.0 -1.15 0.0 0.0
speaker -.92 1.0 0.0 -.92 0.0 0.0
speaker -.69 1.0 0.0 -.69 0.0 0.0
speaker -.46 1.0 0.0 -.46 0.0 0.0
speaker -.23 1.0 0.0 -.23 0.0 0.0
speaker .0 1.0 0.0 .0 0.0 0.0