Synchronized Structured Sound
Real-Time 3-Dimensional Audio Rendering

by

Araz Vartan Inguilizian

Bachelor of Science in Electrical and Computer Engineering,
Bachelor of Art in Art and Art History,
Rice University, Houston, Texas, May 1993

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE in

Media Arts and Sciences

at the

Massachusetts Institute of Technology
September 1995

Author [illegible]
Program in Media Arts and Sciences
August 11, 1995

Certified by

V. Michael Bove, Jr.
Associate Professor of Media Technology,
Program in Media Arts and Sciences
Thesis Supervisor

Accepted by

Stephen A. Benton
Chair, Department Committee on Graduate Students
Program in Media Arts and Sciences

Synchronized Structured Sound
Real-Time 3-Dimensional Audio Rendering

by

Araz Vartan Inguilizian

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
on August 11th, 1995
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE in Media Arts and Sciences

Abstract

Structured Sound describes a synthetic audio environment where sounds are represented by independent audio sources localized in time and three-dimensional space within an acoustic environment. A visual analog, structured video, represents image sequences as a composition of visual components, whose dynamics are controlled by a scripting language and which is rendered/decoded in real-time according to an interactively modifiable viewer location. This research takes the audio components of a script and interactively renders them with respect to the position of the listener/viewer. The audio components are discrete sounds and effects synchronized with the actions of script objects, while the acoustic modeling and processing performed accounts for the listener location within the script "world". Coupled with an interactive scripting language and a structured video system already developed, this work produces a real-time three-dimensional structured audio/video system.

Thesis Supervisor: V. Michael Bove, Jr.
Associate Professor of Media Technology,
Program in Media Arts and Sciences

Support for this work was provided by the Television of Tomorrow consortium.

Synchronized Structured Sound
Real-Time 3-Dimensional Audio Rendering

by

Araz Vartan Inguilizian

Readers

The following people served as readers for this thesis:

Reader

[illegible]
Glorianna Davenport
Associate Professor of Media Technology
Program in Media Arts and Sciences

Reader

[illegible]
Barry Lloyd Vercoe
Professor of Media Arts and Sciences
Program in Media Arts and Sciences

Acknowledgments

This thesis has been a testimony to the abounding grace of our Lord Jesus to me. I was expecting to spend long long hours in the lab, not sleeping and such. But God has blessed me abundantly. Everyday I would spend time praying and then coming to work to complete all that He set aside for me to do. Step by step things came into place, by His grace. And now, inretrospect, I can say that it was fun, because God was with me throughout the whole process. He did not limit his abundance, but also gave me great earthy support. I am thankful to Him for them. Therefore I would like to express my gratitude to:

My advisor, *Mike*, for all the encouragement and help he gave me through this thesis, for keeping the vision going.

My two readers, *Glorianna and Barry*, for their helpful comments and their desire to see something glorious come out of this.

Bill Gardner, for his patience with me as I came up to him seeking help to understand sound, and implement his techniques for this thesis.

Stefan, for making my life easier by writing a scripting language that could use my system, and for being such a good sport.

Linda Peterson, for always comforting me when I came to her seeking help, and for always coming up with a solution.

Shawn and Wad, for spending the time with me patiently explaining how things worked, and never complaining whenever I came by.

David Tames, Katy, the Struct-O-Vision gang and the Cheops gang, for keeping the rest of the system running and making it possible for me to do this thesis.

Lena and Santina, for being the coolest assistants ever.

Henry and the Garden gang, for keeping the garden ship-shape and ready.

The rest of the Garden - especially *Bruce, Brett, Chris, Ed Acosta, Ed Chalom, Frank, Jill, Karrie, Ken, Klee, Nuno, Pasc, Roger* - for your friendships and making the garden a special and fun place to be.

Wole, for being such a close friend, I thank God for you, brother.

Rob and the C-flat gang, for feeding me and blessing me abundantly.

Eliza, for just being the blessed person you are, for sharing God's glory to everyone, and for encouraging me abundantly.

Folusho, for always encouraging me to have faith and to take God at his word.

The rest of Maranatha - *Durodami, Kamel, Ien, Joe, Tope, Rogelio, Suzie, Ale and even Betty*, for your helpful support during this time.

Mary & Mary, for your friendship, encouragement and help with my thesis and with my driving, respectively.

74, for your encouraging and uplifting emails.

Marie Norman and the Thursday night prayer gang, for all the prayer support and the exciting time of faith and deliverance.

Ann Yen, Ashley, Betty, Brian Diver, Derrick, Gaby, Lisa, Patrick Kwon, Patrick Piccione, Santia, and all the rest of my friends who prayed for me and blessed me.

Tree of Life City Church & Framingham Vineyard Christian Fellowship, for being so on fire for Jesus that I could neither escape, not help but be on fire for Him as well.

My family - *Vartan, Takouhie, Taline, Haig and Ani* - for your prayers and support. I love you all.

This is the scripture God gave me regarding this thesis:

"I will make rivers flow on barren heights,
and springs within the valleys,
I will turn the desert into pools of water,
and the parched ground into springs.
I will put in the desert the cedar and the acacia,
the myrtle and the olive.
I will set pines in the wasteland,
the fir and the cypress together,
so that people may see and know,
may consider and understand,
that the hand of the Lord has done this,
that the Holy One of Israel has created it."
(Isaiah 41:18-20)

Thank you Jesus for everything.
Thank you, everyone, for making this thesis possible,
and remember H.I.C. - He Is Coming............

Synchronized Structured Sound
Real-Time 3-Dimensional Audio Rendering

Table of Contents

List of Figures

1 Introduction

When movies were first introduced, people were amazed. No real actors were present, just their images which moved and interacted on screen in a realistic manner. Society quickly learned to accept these moving images as part of their life, but the audio track was limited to an orchestral score accompanying the film. The second wave of motion picture technology incorporated synchronous dialogue and special effects. In this era the audio track was not solely comprised of the score, it also included sound that was related to the objects and action on the screen. The sound was not of good quality, but it was relevant.

The motion picture industry has gone through many other innovations. Many of these innovations dealt with what is seen, but some enhanced what is heard. For example, the introduction of Dolby Surround Sound [Dolby] was a great addition to the movie industry; sound could now be perceived as coming from all around the room and not just from between two speakers. Because of the contributions of object relevant sound, the experience of today's audience is more realistic and enjoyable.

At the Media Laboratory, researchers have abandoned the traditional view that movies are made up of eternal frames. Instead they are moving towards a moving image system in which objects are assembled at the display [Bove 93]. This helps in many ways: one of which is that the system is not bound to an eternal frame, but rather has the flexibility of changing how the movie is viewed by having a few parameters changed. Thus the viewer or a knowledgeable machine is able to interact with the movie. As one interacts with the system and the state of the system is changed, the output changes in response. The viewer is transformed from a passive agent, to an active agent who can customize the sequence and position of viewing according to his preferences.

This thesis will create a structured auditory system as others have built a structured visual system [Granger 95]. The aim is to add realism to the structured video domain by adding synchronous structured sound. This thesis will consider sound as an entity of a structured object, in the same way that a set of frames in a video shot are considered a separate entity. This sound entity has the ability to be localized in 3-D space to support the visual 3-D of the video structured system. These localizations have the capability of changing instantly according to information received about their new location. The system adds an acoustic environment that corresponds to the visual virtual room the viewer is in.

1.1 A Generic Structured System

In a general structured system, all components of the final product can come from any system and are composed together dynamically in real-time. In such a system, the script and not the frames defines the final product. In a traditional system, a movie is produced from a script which predetermines not only the plot but also the sequence of events and views. However in a structured movie, this is not the case, the script establishes the guide lines of the plot, but the sequence of events and views has a dynamic capability of changing with setup and user input. What the audience sees depends on the system setup and the audience's interaction with the system. Therefore every aspect of the production system has to be geared to produce adaptive elements that can be composed in real-time from raw data.

The major components of a structured system so far are the audio and the video components. The audio has to handle interactive sound localization, as well as virtual atmosphere creation. The video has to handle interactive 3-D view change, as well as the compositing of multi-layer images. The scripting language has to have the ability to adapt from one view to another depending on user input. In the future, new dimensions, such as wind or moving seats, could be added along side audio and video.

It is highly unlikely for one machine to be able to do all the work necessary for a production of such magnitude. Thus a modular system is proposed to achieve this (Figure 1-1). The best system is found for each dimension, and they are put together to achieve the final result. The scripting language also has to be modular in form because it has to be able to handle the different aspects of each dimension effectively.

In the system setup at the Media Laboratory, the Video Decoder is the Cheops imaging system(described in more detail in chapter 2). Cheops has the ability to compose 2-D, 2 $^1/_2$-D, and 3-D objects in real-time. The audio decoder is SSSound running on a DEC Alpha 3000/300 running under OSF 3.0. Both the video decoder and the audio decoder receive their information from a local database. Cheops has the ability to store up to 32 Gigs of data in RAM, thereby speeding up the fetch cycle of a raw video data request.

Most of the information stored in this database comes from Sequence and Shot Design. This art has not been perfected yet since the technology has only recently come to the attention of the motion picture industry. Somehow one must shoot the necessary raw data so that all reasonable and possible dimensions of the movie can be available for the user to interact with. Once these shots are recorded and placed in the database, an intelligent script must be written to provide the backbone of the movie. The script is fed into an interpreter along with the system setup, and the dynamic user input. The interpreter is responsible for sending the appropriate instructions to each of the decoders. The Video Decoder would receive an instruction such as "Render background 'a' from viewpoint 'b' and compose actor frame number 'c' in position 'd' in the frame", while the Audio Decoder would get an instruction such as "Start playing sound 'a' at position 'd', starting at time 'b', using parameters of the room 'c'".

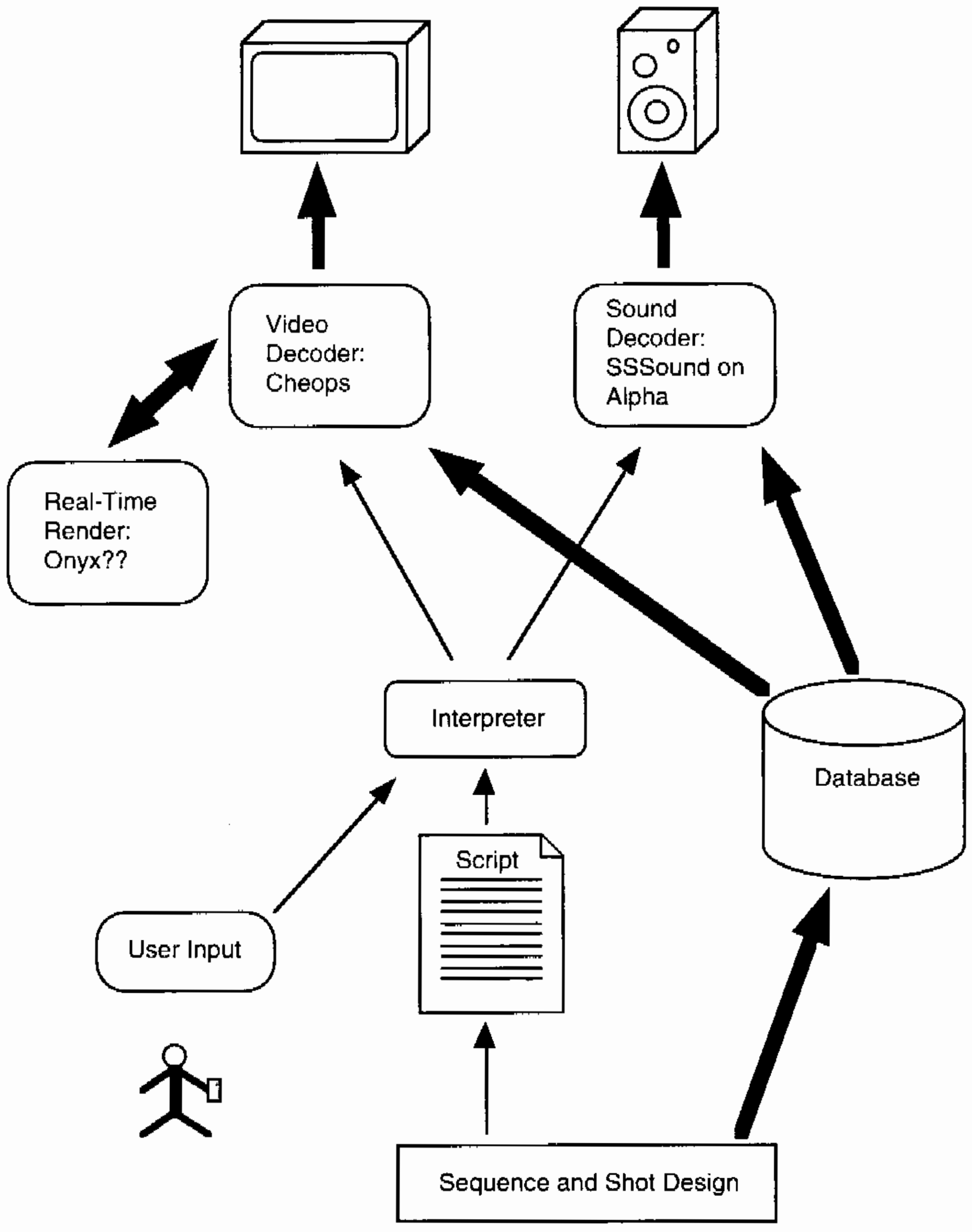

Figure 1-1: A Generic Structured System. The original data is first sorted and stored in a database. Then a sequence shot is transformed to a script and fed into and interpreter which Here the bold arrows represents the data pathways, while the narrow arrows are control instruction data paths.

1.2 Thesis Overview

This thesis will produce a system setup such as Figure 1-2. Here the audience is in the middle of the actual room interacting with the system using a user-input. The screen is a projected screen of the Cheops Imaging system, while the two modeled speakers are receiving audio samples from an Alpha LoFi card. In the original model, there are supposed to be 6 speakers, however, because of the constraints of the system, only two speakers can be modeled(more details in section 4.1.1). The system can handle more speakers, with a maximum of two speakers per Alpha machine.

This thesis will present in Chapter 2 an overview of the Video Decoder used in the Media Laboratory. Chapter 3 outlines the techniques of the Sound Localization methods used in placing the sound. Then Chapter 4 discusses the details of the Audio Decoder, with particular emphasis on the specific constraints of the system used in the Media Laboratory. Chapter 5 describes the external functions available for systems to communicate with SSSound, and it goes into some detail on the implementation of the Isis script interpreter currently in development at the laboratory. Finally Chapter 6 outlines the results and conclusions of this thesis.

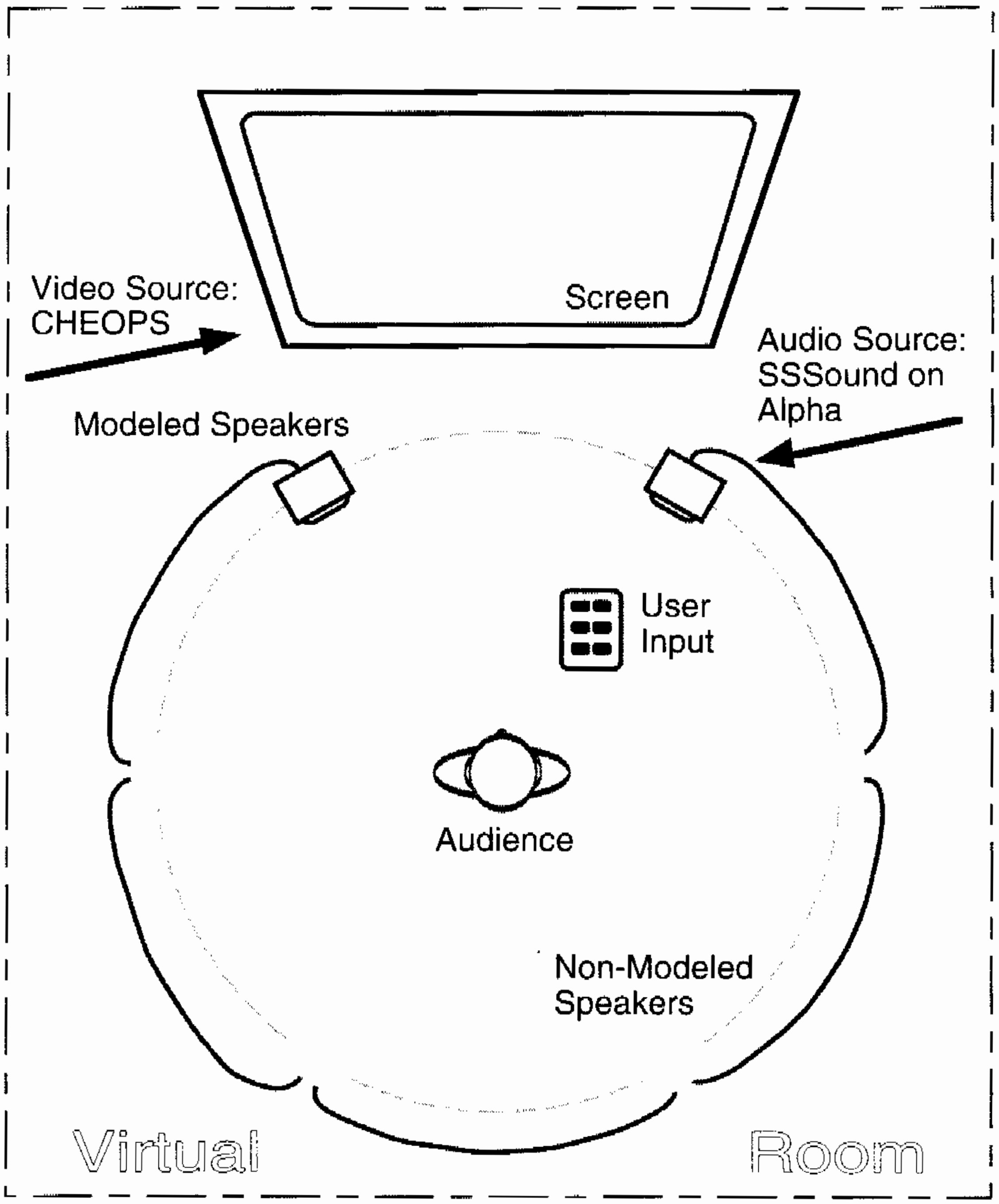

Figure 1-2: A structured imaging system virtual space. Here the video inputs are received from the Cheops imaging system, while the audio inputs are from an Alpha machine running SSSound. Note: only two speakers are modeled because of system constraints though more could be easily added.

S o u n d

2 Video Decoder

The initial breakthroughs in structured systems have been in the video domain. A structured video is a representation of a movie or moving image made up of raw building blocks composed together at the user end in real-time. These raw building blocks are derived from the current system of viewing moving images. They come in three levels of complexity. These objects can be frames with constant depth (2-D), surface maps of objects from particular view points(2 $^1/_2$-D), computer graphics objects (3-D), or layered objects with associated intensity, velocity and opacity maps (2-D or 2 $^1/_2$ D). Objects in a structured system are assembled at the user end using a scripting language which defines every object's location and orientation at every point in time. The system needs great flexibility to manipulate all these types of raw data together into one uniform output. For example, the system should be able to place a moving 2-D image of an actor in a 3-D schematic of a room and to view it from different locations in the room at different times.

2.1 The pipeline

A structured system should have the ability to process 2-D, 2 $^1/_2$-D, and 3-D objects and to present one output. This process can be difficult because each of the objects needs a different form of data manipulation. For example, a 3-D object such as a particle database would need rendering to a specific viewpoint and scaling to the appropriate size, while a 2-D, or 2 $^1/_2$-D object might need scaling and/or warping. A generic structured video decoding pipeline[Bove 94] is suggested in Figure 2-1. Each of the object classes can be manipulated by the system and other object types can be added as they arise. It is assumed that there is a higher process controller that decides what to do with each object in the system, and gives appropriate instruction to the elements in the pipeline at appropriate time. This higher

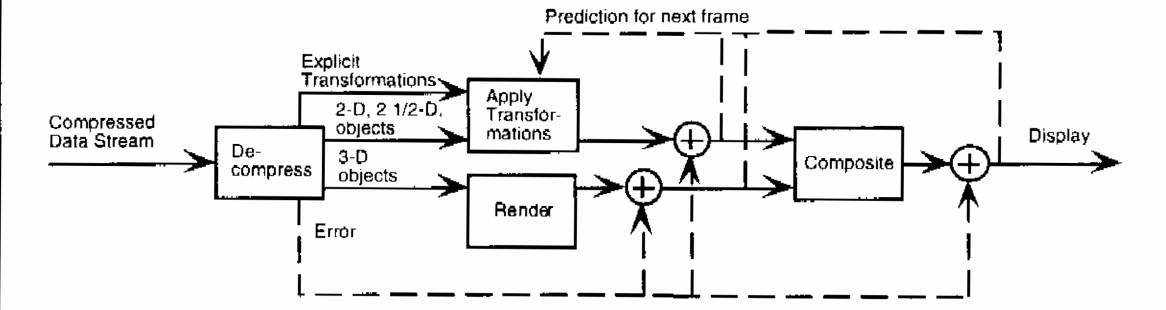

Figure 2-1: Structured video display processing pipeline This pipeline handle all the hard-core processing, while the scripting language(not shown) gives the instructions as to when/where/how each object is.

process controller can be thought of as the bounding language, such as a scripting language as described in section 5.2. It is at this higher level of abstraction that interactivity can be introduced. This allows other parts of the structured system, such as the audio decoder, to respond to the interactivity in an appropriate manner. All the decisions are made at the scripting level, while the hard work is done at the composing pipeline level.

Each of the different object require different data manipulation.

- *2-D objects:* The raw data is arrays of pixels, which the scripting language would assign a depth value to, to facilitate in layering. They may have transparent holes through which other object can be seen through these holes.
- *2 $^1/_2$-D Objects:* These are 2-D objects with depth associated to every pixel. These depths are stored in what is called a *z-buffer*. 2-D objects become 2 $^1/_2$-D objects after the script adds a constant depth value to them, while a 3-D objects become a 2 $^1/_2$-D objects after being rendered from a particular camera viewpoint.
- *3-D Objects:* These are regular 3-D computer graphic representations of objects, ranging from particle databases to texture mapping. Particle databases are used in the Cheops system (described in the next section) because of hardware design features.
- *explicit transformations:* This is an explicit spatial transformation of pixels, otherwise referred to as warp. These may be of different forms such as a dense optical flow field, or a sparse set of motion vectors.

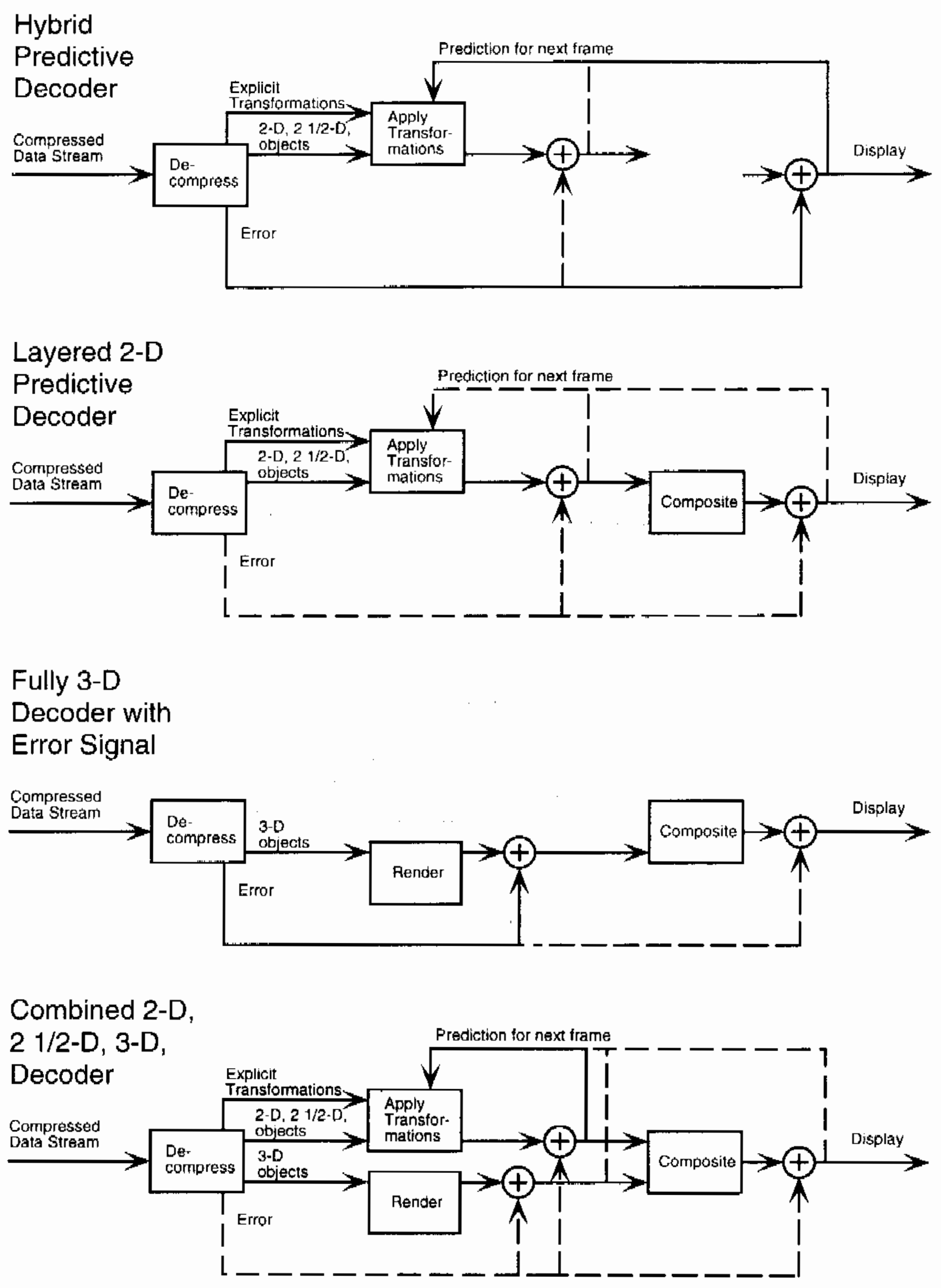

Figure 2-2: The video decoder pipeline showing a variety of possibilities of decoding different forms of structured video objects. The Gray data paths are inactive, while dashed data paths interconnect in an algorithm-dependent manner.

- *error signal:* These are 2D arrays of pixel values which could be added to the result of the transformation or rendering, or even to the composite result at the end.

As shown in Figure 2-2, different structured video objects follow different paths through the pipeline. Therefore, many types of decoders could be used to compose many objects into one output.

2.2 Cheops: An implementation

The Television of Tomorrow Group at the Media Laboratory has been very active in the research of new concepts of video acquisition, processing and display. They have developed the Cheops Image Processing System for this very purpose. Cheops has real-time capabilities to scale, warp and composite 2-D, 2 $^1/_2$D and rendered 3-D objects. It also has the capability to hold up to 32 Gigs of RAM, which can store all the raw data needed for most movies. Cheops is modular in form, in that it divides up the work into small and computationally intensive stream operations that may be performed in parallel and embodies them in specialized hardware.

Using Cheops, Brett Granger developed a system to decode and display structured video objects in real-time[Granger 92]. This system was used to decode and display a structured movie, named *The Museum*, developed by the Television of Tomorrow Group of the Media Laboratory. This movie was shot and developed with the purpose of showing flexibility and user interactivity. The plot is as follows: A man walks into a museum where in the middle of the room is a statue and a frame in front of the statue. The man walks around the statue and is intrigued by it. He finally looks through the frame and the statue becomes alive and motions the man to come forward. Once the man responds, he turns into a statue, and the statue becomes a man and walks out of the scene.

The acquisition of the raw data for this movie was done by two methods. The first model is the background. From three normal pictures of a museum, a 3-D model of the gallery was extracted using Shawn Becker's semiautomatic 3-D model extraction technique

[Becker 95]. Then, using texture mapping, the frame was added into the middle of the room as part of the room. The second model was the actors. Here the actors were video taped from three directions in front of a blue screen performing the actions in the script. The actors were extracted out into a set of 2-D objects. These two object types are the building block of the movie, along with the scripting language.

The scripting language allows the users to control the location of an object, scale, frame and view position. The location of the objects could either be a 2-D location on the screen (in which case scale would be needed to ensure relative height), or a 3-D location in the simulated space (where scale is not needed since it can be calculated from the camera parameters and the 3-D location). Frame refers to the temporal frame number from the sequence of video still captured on tape. Since there might be more than one recording of an event from different locations, the view position is used to control which one of the multiple views is needed depending on the current view direction. The scripting language also controls the view parameters, which are the location and direction of the camera, the focal length and the effective screen dimensions.

All the above parameters could be dynamically controlled by the script, or even placed under user control. This allows flexibility in the viewing of the movie. For example, the system could change the location of the camera by simply turning a knob (as user input).

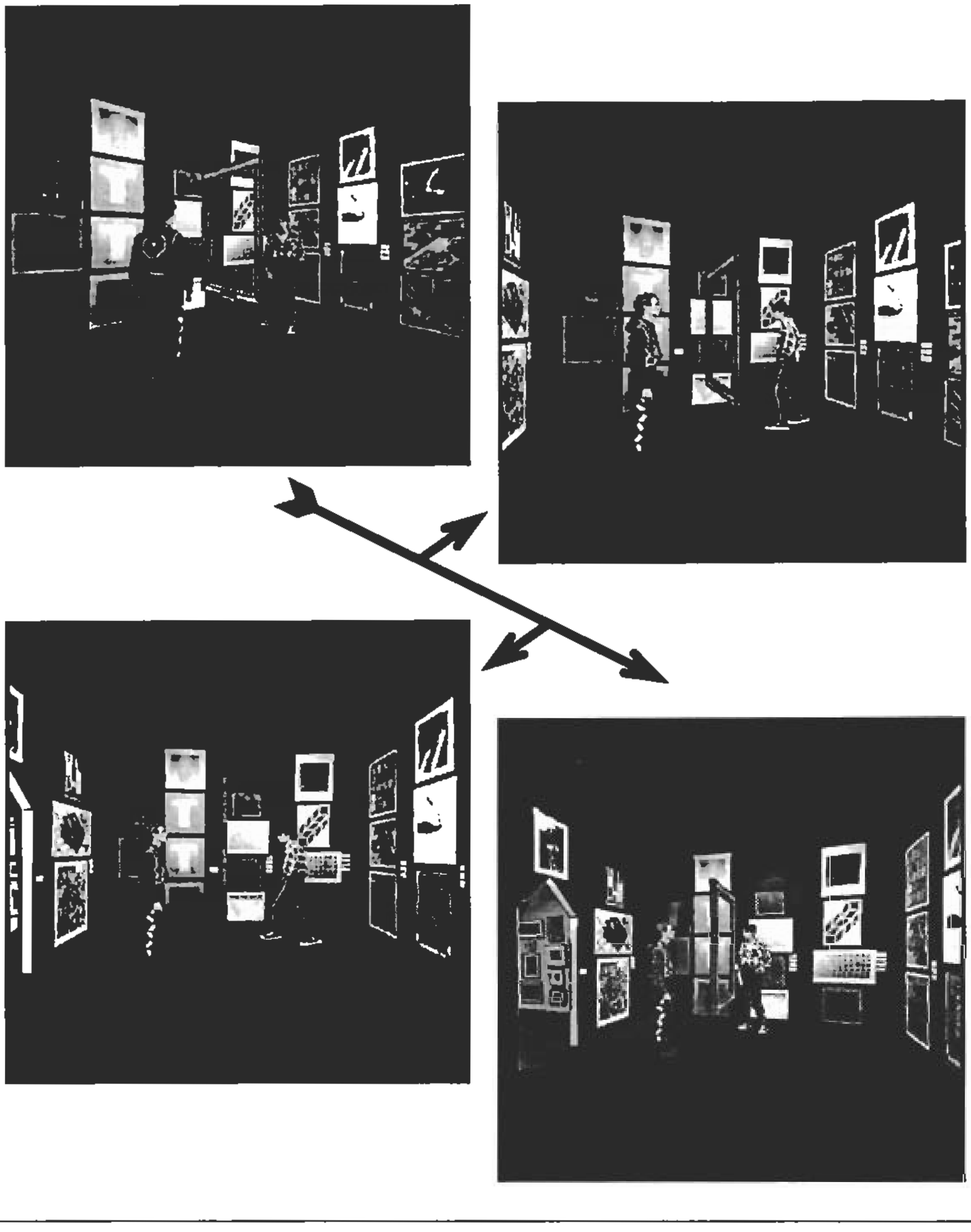

Figure 2-3: A rotation in *The Museum* movie. Here the user is requesting a rotation along an axis, and thereby the output is four shots of a rotated view of the room.

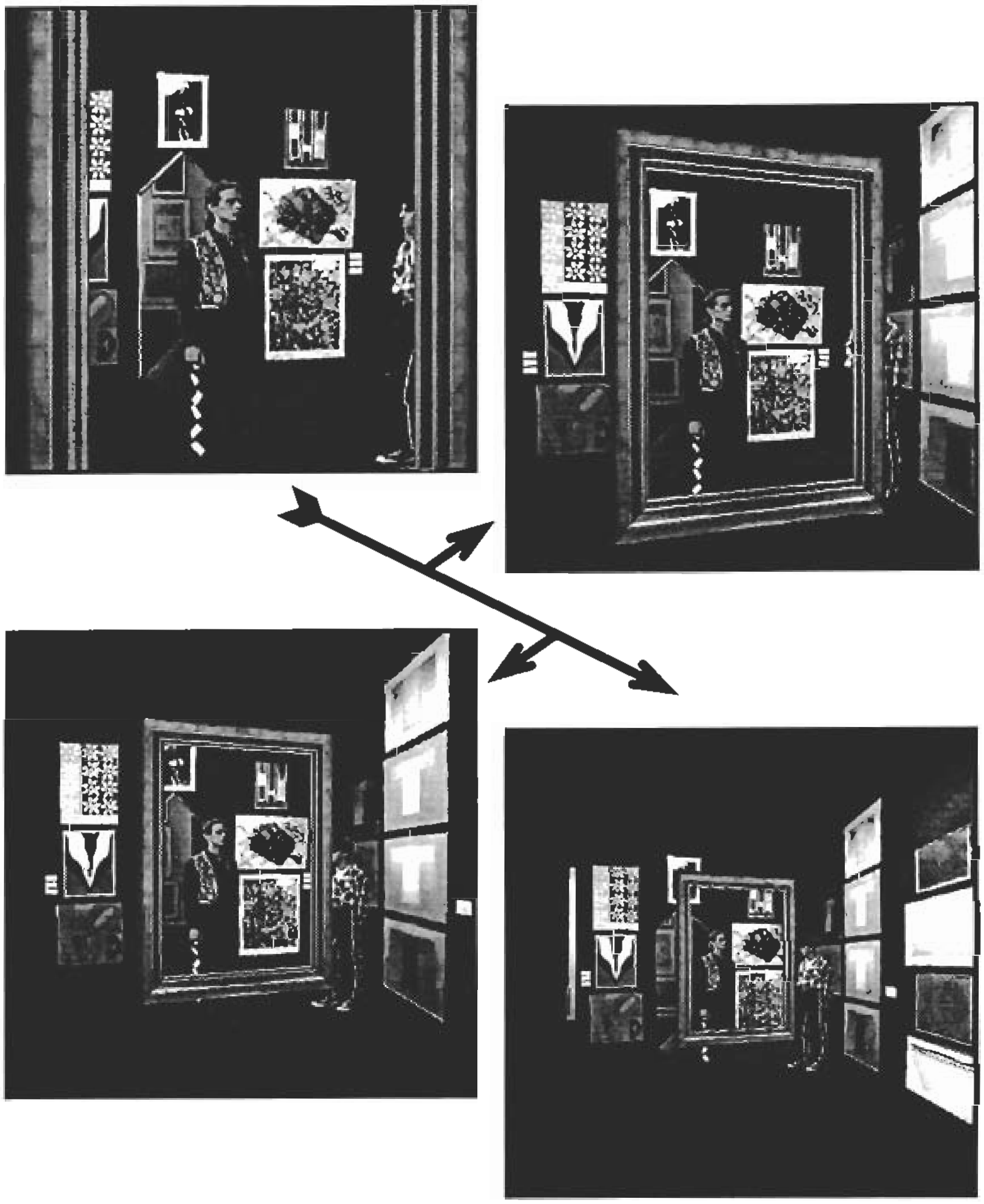
Figure 2-4: A zoom in *The Museum* movie. Here the user is requesting a zoom out, and the output is four shots of a zoom out of the room.

3 Sound Localization

This thesis concentrates on the idea of Structured Sound. A system such as this requires a mode of delivering the sound in real-time so as to appear that it is coming from a certain location. If an image appears on the left side of the room two meters in front of the audience, so should the sound appear to be coming from the left side of the room, two meters in front of the audience. This procedure is called localization of sound.

Engineers for years have tried to design a system to synthesize directional sound. The research in this field is split between two methods of delivering such directional sound. The first uses binaural cues delivered at the ears to synthesize sound localization. While the second uses spatial separation of speakers to deliver localized sound. A binaural cue is a cue which relies on the fact that a listener hears sound from two different locations - namely the ears. Localization cues at low frequencies are given by interaural phase differences, where the phase difference of the signals heard at the two ears is an indication of the location of the sound source. At frequencies where the wavelength is shorter than the ear separation, phase cues cannot be used; interaural intensity difference cues are used, since the human head absorbs high frequencies. Thereby, using the knowledge of these cues and a model of the head, a system can be implemented to give an illusion of sounds being produced at a certain location. Head Related Transfer Functions (HRTF) [Wenzel 92] have been used extensively to deliver localized sound through head-phones. The HRTFs are dependent on ear separation, the shape of the head, and the shape of pinna. For each head description, the HRTF system produces accurate sound localization using headphones. However, because of the system's dependence on ear separation, no two people can hear the same audio stream and "feel" the objects at the same location. There would need to be a different audio stream for each listener depending on his/her head description.

One of the requirements of this thesis was to produce sound localization for a group of people simultaneously. Therefore, the author could not use binaural cues to localize sound sources. The second method of sound localization was used, that is the delivery of localization cues using the spatial distribution of speakers. Here the system does not rely on the fact that the audience has two ears, rather the system relies on intensity panning between adjacent speakers to deliver the cue. Thus an approach developed by Bill Gardner of the Media Laboratory's Perceptual Computing Group was adopted. Gardner developed a design for a virtual acoustic room [Gardner 92] using 6 speakers and a Motorola 56001 digital signal processor for each speaker, on a Macintosh platform.

This work is not identical to Gardner's. Because of design constraints, the author was limited to Digital's Alpha platform, and to the LoFi card [Levergood 9/93] as the primary means of delivering CD-quality sound. The LoFi card contains two 8 KHz telephone quality CODECs, a Motorola 56001 DSP chip with 32K 24-bit words of memory shared with the host processor. The 56001 serial port supports a 44.1 KHz stereo DAC. Thus using only one Alpha with one sound card limited the number of speakers to be used to two[1]. Those two speakers would have to be placed in front of the listener, which might limit the locations of sound localization. Gardner's model assumes that the listener is in the middle of the room and that sounds could be localized anywhere around him; he uses 6 speakers spaced equally around the listener for this purpose. However, from a visual standpoint, people view three dimensional movies through a window, namely the screen. The audience is always on one side of the window and can only see what is happening through that window. Therefore, it may not be too much of a restriction to limit sound localization to the front of the audience. The system would have two real speakers and one virtual speaker behind the listener that is not processed, but is needed to ensure proper calculations.

There are two major processes that are executed to render sound localization from the spatial location of speakers. The first is the simulation of the early reverberation response hereafter referred to as the Echo process. This is the modeling of the direct reflections from the walls of a room. The Echo process will produce an FIR filter for every

[1] More LoFi cards could have been placed in the system, however the processing required for two speakers takes up all of Alpha's processing power. The sampling rate could have been halfed, to accomodate two other speakers; however, the author decided to keep to the full 44.1KHz for clarity in the high frequencies.

speaker representing the delay of all the virtual sources in the room. The second is the simulation of the late reverberation, or diffuse reverberation, hereafter referred to as the Reverberation process. This models the steady state room noise from all the noises in a room and their Echoes. It is not directional such as the Echo is, but rather it creates the general feel of the acoustic quality of a room.

3.1 The Echo Process

The echo process is an attempt at simulating all the virtual sound sources resulting from the sounds bouncing off walls. The echo process is divided up into three procedures. The first is for every sound source in the room, to calculate the virtual sources beyond the room because of the reflections off the walls. The second is for every sound source and speaker, to calculate an FIR filter representing the delay of the virtual sources that are needed to be projected from that speaker. The third is the pruning stage for real-time purposes, where the FIR filter is pruned to reduce the number of taps, and thereby calculations, needed in the real-time rendering of the sound streams.

3.1.1 Virtual Sources

The first stage of the Echo process is to compute all the virtual sources in the room for every sound stream in the room. Since this system is confined to rectangular rooms (see section 4.1.2 for more detail), the procedure is simple (Figure 3-1). The program loops through the number of reflections, from zero to the user defined max_reflections for the room. For every number of reflection, the program loops through every possible reflection on the x-axis walls, and calculates the z-axis wall reflections necessary. Once the program has defined x-axis and z-axis locations, then it needs to calculate the attenuation coefficient for every location due to the reflections off the walls. The result depends on the number rather than the order of reflections on each wall. To calculate the number of reflections on each wall, the program would divide the axis location by two and the whole number result is that number of paired reflections on both walls on the axis, and the remainder is used as an indicator of any single reflection apart from the pair reflections. The sign of the remainder defines which wall the

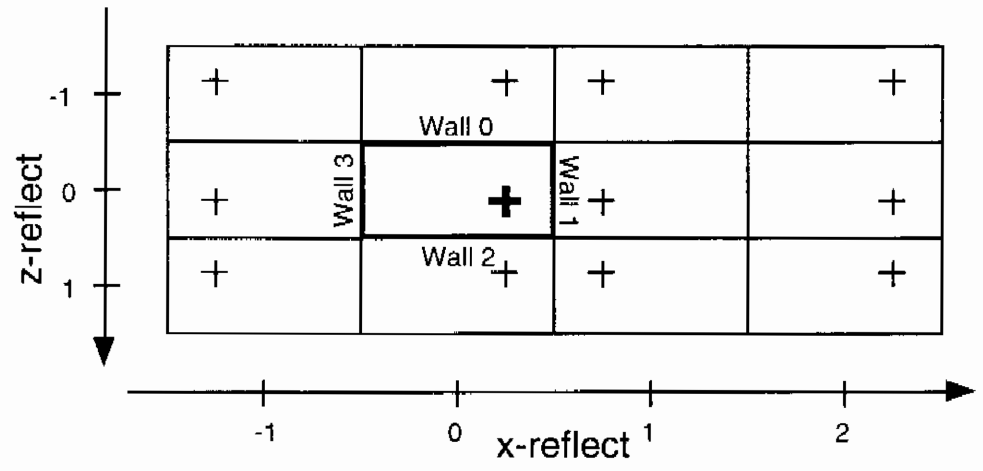

Figure 3-1: Echo effects of sources in a rectangular room. The BOLD lines represent the actual room and source location, while the normal lines represented the echoed rooms/source locations. The axis is the number of reflections on the perpendicular walls. For example, x=3 means 2 reflections on wall 2 and 1 on wall 4. NOTE: The walls are marked in clockwise order starting from minus-z location, and the listener is facing the minus-z direction.

reflections occurred; for example, remainder of x = -1 would mean a bounce off wall 3 (the minus-x wall) and not wall 1(the plus-x wall). As a general example assume the program is calculating 11 reflections, and it happens to be on x-reflect-axis -5, and z-reflect-axis +6. The program computes the following:

X-reflect-axis

$$\begin{tikzcd}
\frac{-5}{2} = -2 \arrow[d, "\text{Number}"'] & + rem(-1) \arrow[dd, "\text{Sign}"] \\
2 * [ref\_coef(wall1) + ref\_coef(wall3)] & \\
& + ref\_coef(wall3)
\end{tikzcd}$$

Z-reflect-axis

$$\begin{tikzcd}
\frac{+6}{2} = +3 \arrow[d, "\text{Number}"'] & + rem(0) \\
+\{3 * [ref\_coef(wall0) + ref\_coef(wall2)]\} &
\end{tikzcd}$$

The above summation is the attenuation coefficient of the virtual source, where ref_coef is the reflective coefficient of a wall. The attenuation coefficient is multiplied by the tap amplitude of that virtual source.

3.1.2 Speaker Sound

The next step in the process is to create an FIR filter for each speaker, representing the delays from all the sources in the room. From the list of the virtual sources, the program picks out the sources that could be projected from each speaker, and uses intensity panning between adjacent speakers to achieve the desired spatial localization of the virtual sources [Theile 77]. Moreover since the listener is not constrained to any particular orientation, it is unclear how to use phase information to aid in the localization of sound.

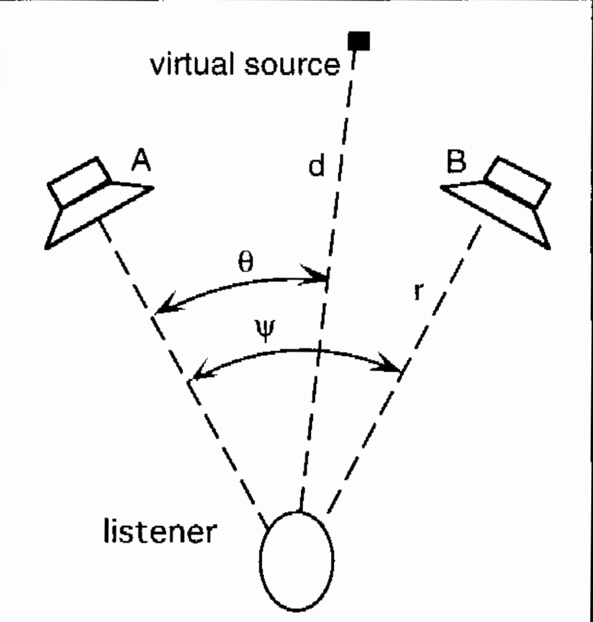

Figure 3-2: Intensity panning between adjacent speakers.

The diagram on the right (Figure 3-2) depicts one of the virtual sources in the system between two speakers. This virtual source will contribute a tap delay to both the speakers A and B, but not to any other speaker. The tap delays are proportional to the difference of the distances from the listener to the speaker and to the virtual source. The tap amplitudes are dependent on the same distances as well as the angle spans.

The formula for this system is as follows:

$$\text{A, B tap delays} = \frac{d-r}{c}$$

$$\text{A tap amplitude} = a\frac{r}{d}\cos\left(\frac{\pi\theta}{2\psi}\right)$$

$$\text{B tap amplitude} = a\frac{r}{d}\sin\left(\frac{\pi\theta}{2\psi}\right)$$

$$a = \prod_{j\in S}\Gamma_j$$

where :

d is the distance to the source in meters,

r is the distance to the speakers in meters,

c is the speed of sound in meter per second,

a is the amplitude of the virtual source relative to the direct sound,

S is the set of walls that sound encounters, and

Γ_j is the reflection coefficient of the jth wall.

There are a couple of comments that are worthy to be noted:

- The value of a was calculated when the program found each echoed source, and it was stored in the sound source description.
- This result assumes that the listener, speaker and virtual sources all lie in the same horizontal plane, and the speakers are all equidistant from the listener.
- The speaker locations are fixed with respect to the front of the listener. Therefore if the listener is facing a direction other than minus-z in the virtual space, then the speaker locations need to be rotated by that same amount and direction before any of the above calculations could be performed.

3.1.3. Pruning the Filters

A typical system setup of a rectangular room might have the maximum reflections of the room set to eight. This would give us 64 filter taps. While there is no direct system limit on the number of taps, the more taps the filter has the longer the program would take to compute the result of the filter over the sound samples. In a real-time environment, every possible care should be taken to force the system to compute a reasonable result as fast as possible. Therefore to enhance real-time performance, the procedure used to intelligently reduce the number of filter taps is as follows:

- Adjacent filter taps within 1 millisecond of each other are merged to form a new tap with the same energy. If the original taps are at times t_0 and t_1, with amplitudes a_0 and a_1, the merged tap is created at time t_2 with amplitude a_2 as follows:

$$t_2 = \frac{t_0 a_0^2 + t_1 a_1^2}{a_0^2 + a_1^2} \qquad a_2 = \sqrt{a_0^2 + a_1^2}$$

- Filter taps are then sorted by amplitude. A system defined number of the highest amplitude taps are kept.

The pruning process tends to eliminate distant virtual sources as well as weak taps resulting from panning. This process should not affect the system quality if the maximum filter taps is set to at least 50 or so. The higher the max-number-taps, the better the system quality. However if real-time performance is hampered, lower max-number-taps would be advised.

3.2 The Reverberation Process

Rooms do not produce just direct reflection off of walls, they also have a general steady state noise level from all the sounds produced in the room. This noise is a general feeling of the acoustic quality of the room, and is referred to as Reverberation. Rendering this reverberant response is a task that has confounded engineers for a long time. It has been the general conception that if the impulse response of a room is known, then the user can compute the reverberation from many sound sources in that room. A system that Moorer determined to be an effective sounding impulse response of a diffuse reverberator is an exponentially decaying noise sequence [Moorer 79]. Rendering this reverberator requires performing large convolutions. At the time of Gardener's system development, the price/performance ratio of DSP chips was judged to be too high to warrant any real-time reverberator system at reasonable cost. Perhaps the ratio is now low enough to allow for real-time reverberator systems at reasonable cost. If a system incorporating these chips is implemented, input would have to be convolved with an actual impulse response of a room, or a simulated response using noise shaping. However, no such DSP chip exists for the Alpha platform. Thus the system implements efficient reverberators for real-time performance. This requires using infinite impulse response filters, such as a comb and allpass filters.

Two considerations were present when choosing which of the many combinations of filters to implement. The first consideration was the stability of the system at all frequencies. The second was that the system would increase the number of echoes generated in response to an impulse, since in a real room echoes, though they subside, increase in number. Thus, nested allpass filters are chosen as the basis for building the reverberator, since they satisfy both the parameters. For more detail on the mathematics and the creation of different reverberator refer to [Gardner 95] and [Gardner 92].

The design of the nested allpass filters used in the system is modeled in Figure 3-3, where X is the input, Y is the output, g is gain and G(z) is simply a delay. This allpass filter is the building block. The result of cascading these filters together is not a good sounding reverberator; it's response is metallic and sharp sounding. However, when some of the output of the cascaded allpass system is fed back to the input through a moderate delay, great sounding reverberators are achieved. The harsh and metallic feel of the systems without the feedback is eliminated partly because of the increased echoes due to the feedback loop. Moreover, adding a lowpass filter to the feedback loop would simulate the lowpass effect of air absorption. This newer system would be of a form as shown in Figure 3-4.

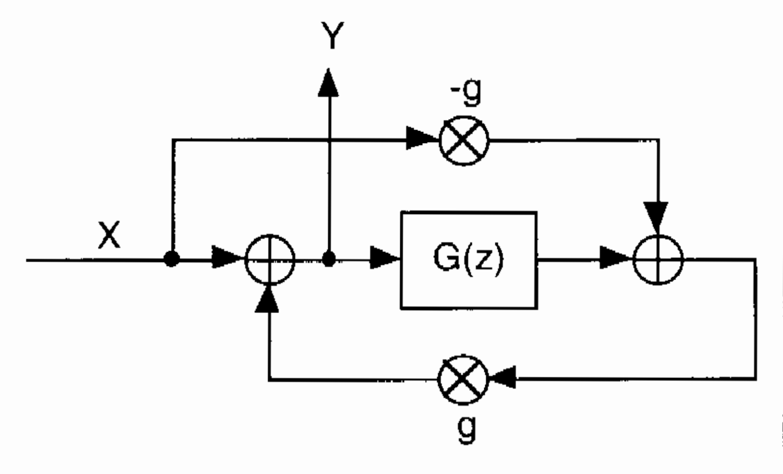

Figure 3-3: Allpass flow diagram with samples taken from the interior of the allpass delay line.

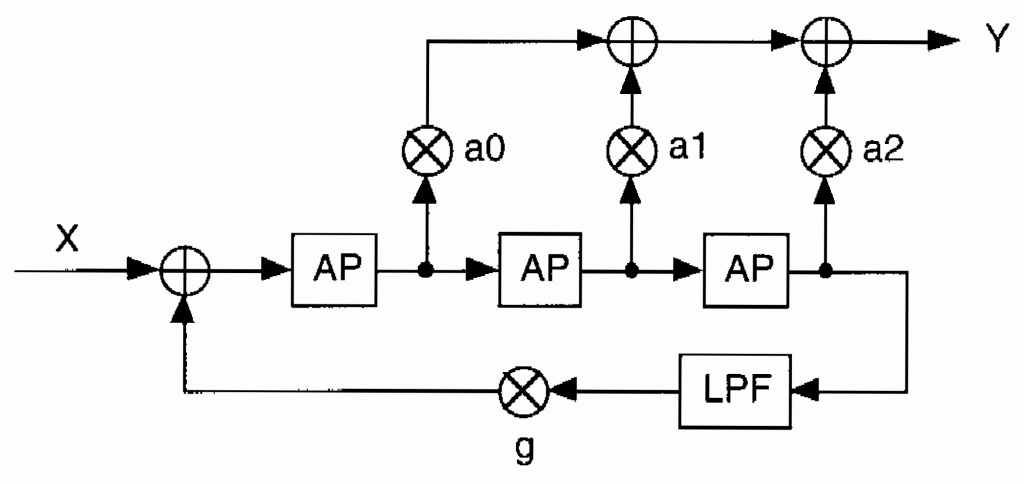

Figure 3-4: A generalized allpass reverberator with a low pass filter feedback loop, with multiple weighted output taps.

The system represents a set of cascaded allpass filters with a feedback loop containing a lowpass filter. The output is taken from a linear combination of the outputs of the individual allpass filters. Each of the individual allpass filters can themselves be a set of cascaded or nested allpass filters. The system as a whole is not allpass, because of the feedback loop and the lowpass filter. Stability would be achieved if the lowpass filter has magnitude less than 1 for all frequencies, and g (gain) < 1.

From this general structure, many systems can be designed. The key to creating good sounding reverberators is not mathematics, but rather it is the ear. The basic decision criterion for finding a good reverberator is whether or not it sounds good. Since the ear is good at detecting patterns, the job of a good reverberator is to elude this pattern recognition process. Therefore the reverberators used in SSSound have been empirically designed to sound good. They are taken from Gardner's Masters thesis[Gardner 92]. None of them are mathematical creations, rather they are the result of laborious hand tweaking, so as to produce good sounding reverberators.

In order to simplify the representation of nested allpass reverberators, a simplified schematic representation was used as shown in Figure 3-5. The top of the figure (3-5a) is the procedure used to perform the allpass filtering. Here the feed-forward multiply accumulate through -g occurs before the feedback calculation. While figure 3-5b shows a simple nested allpass system (for instructional purposes only). The input enters a delay line at the left side, where it is processed with a single allpass followed by a double nested allpass. The allpass delays are measured in milliseconds, while the gains are positioned in parenthesis. The system first experiences a 20 milliseconds delay, then a 30 millisecond allpass with a gain of 0.5. Then the system passes through another 5 milliseconds of delay, followed by a 50 millisecond allpass of gain 0.7 that contains a 20 millisecond allpass of gain 0.1.

In a general reverberator such as the one described in Figure 3-4, the only variable in the system is the gain of the feedback loop. Tweaking this gain would give us different reverberation responses. However, just this variable is not enough to simulate all the sizes of rooms a system can encounter. Thus it is highly unlikely that such a reverberator could

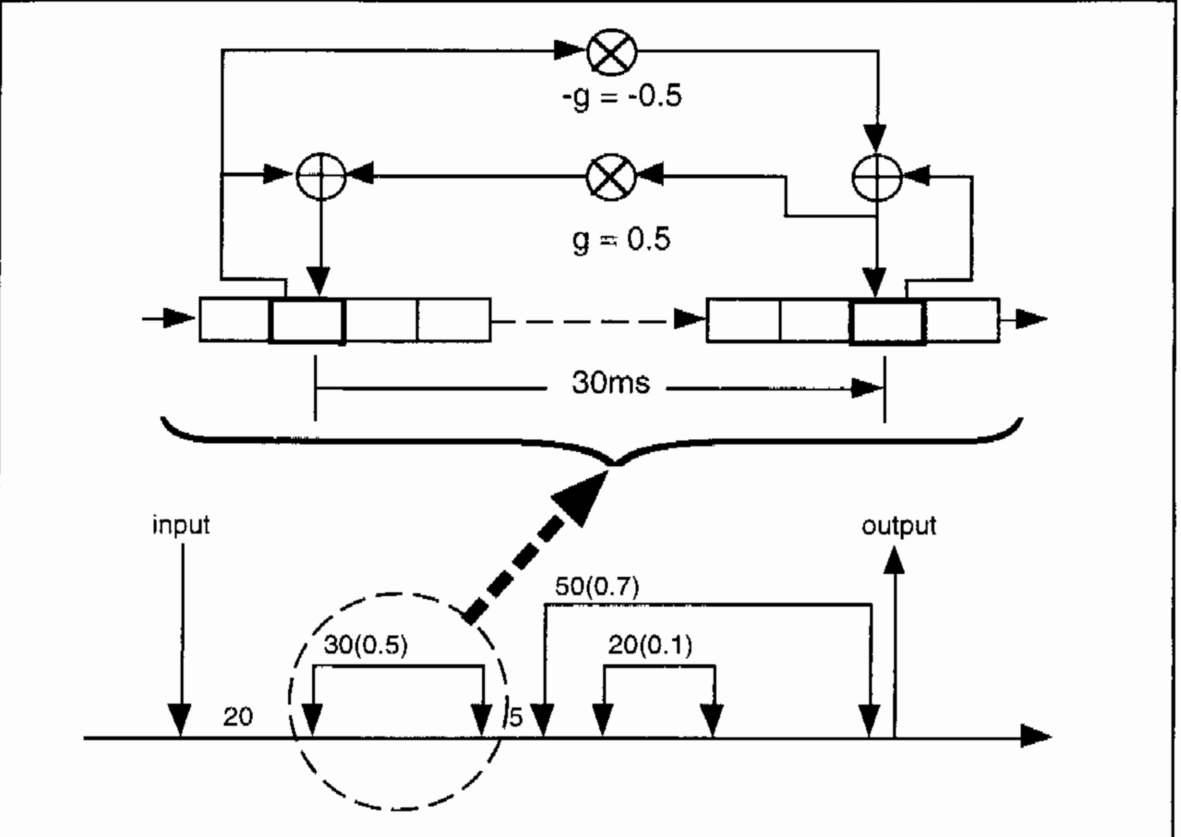

Figure 3-5: A detailed description of the all pass procedure and an example of a reverberator system. **a**) (top) a schematic of the allpass procedure, where the forward multiply accumulate (of -g) happens before the feedback calculation through +g. **b**) (bottom) instructional allpass cascaded system.

be designed to simulate all the types and sizes of rooms. Gardner suggested three reverberators, one for each small, medium and large sized rooms. The acoustic size of the room can be established by the reverberation time of the room. The reverberation time of the room is proportional to the volume of the room and inversely proportional to the average absorption of all the surfaces of the room.

The following formula is a method of calculating the reverberation time (T) of a room:

$$T = \frac{60V}{1.085ca'} = 0.161\frac{V}{a'}$$

$$a' = S[-2.30\log_{10}(1-\overline{\alpha})]$$

where:

T is reverb time in seconds,

c is the speed of sound in meters per second,

V is the volume of the room in meters cubed,

a' is the metric absorption in meters squared,

$\overline{\alpha} = \frac{S_1\alpha_1 + S_2\alpha_2 + \ldots + S_n\alpha_n}{S}$	S is the total surface area in meters squared,
	$\overline{\alpha}$ is the average power absorption of the room,
$S = S_1 + S_2 + \ldots + S_n$	S_i . α_i is the surface area and power absorption of wall i, and
$\alpha = (1 - \Gamma^2)$	Γ is the pressure reflection of a material.

The above formula is used to calculate the reverberation time of the room so as to know which reverberator to use. The following table shows the reverberation time range for each room:

Reverberator	Reverberation Time (sec)
small	0.38 -> 0.57
medium	0.58 -> 1.29
large	1.30 -> infinite

Figure 3-6 shows the three reverberators used in SSSound.

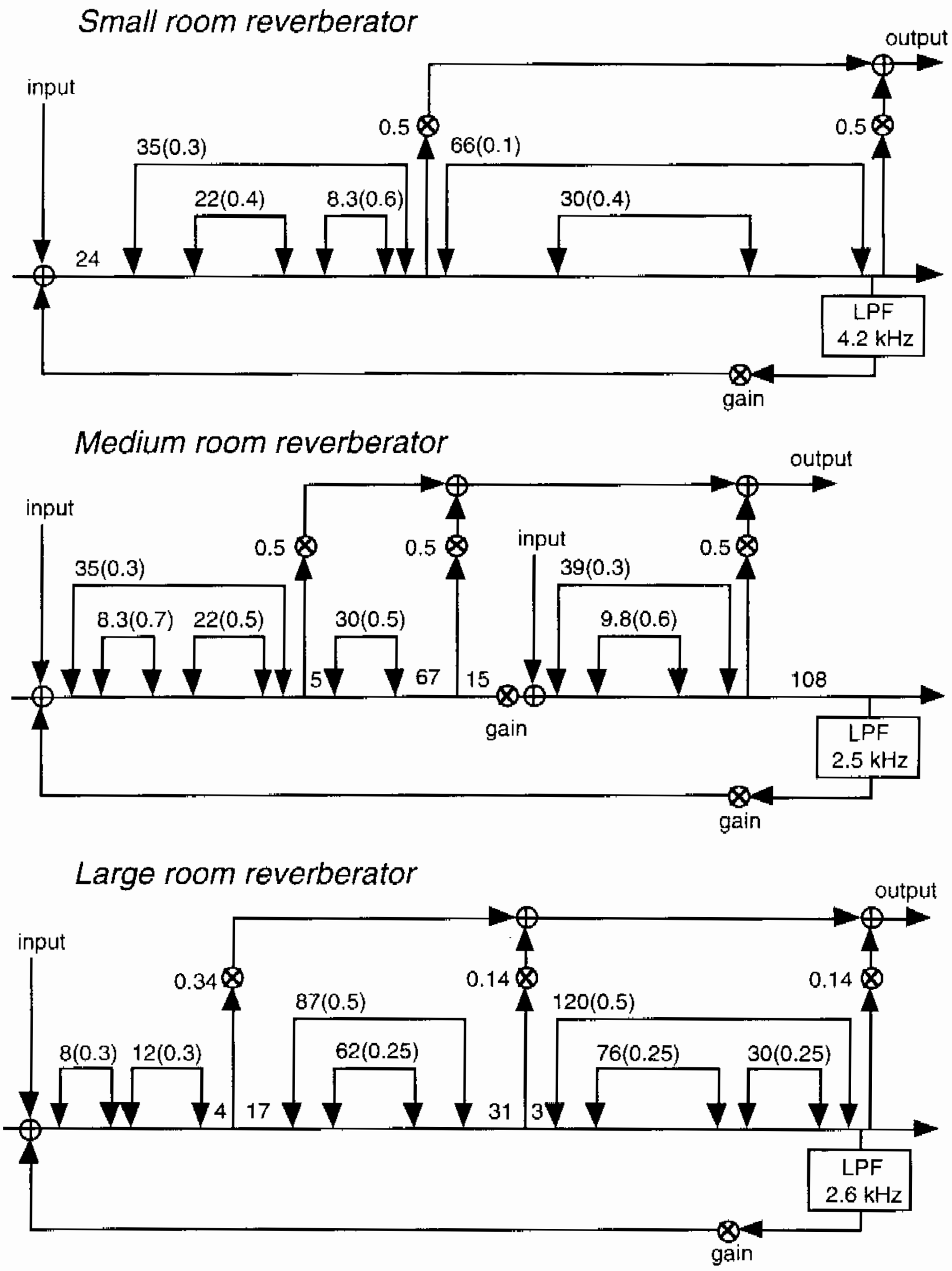

Figure 3-6: Diffuse reverberators used in SSSound for small, medium and large rooms. See figure 4-5 for detailed explanation of the schematic. These reverberators were designed by W. Gardner [Gardner 92].

4
Sound Decoder

The heart of SSSound is the "Engine". The Engine is the part of the code responsible for plugging through the mathematics and computing the final output result. It runs on a three thread system, a Play thread, a Setup thread and a Comm thread (Figure 4-1). The Comm thread gives the instructions to SSSound. It could be a scripting language that has been adapted to run SSSound, or it could be a decoder that receives messages over a socket and runs instructions used by SSSound (in-depth discussion in chapter 5). The Setup thread handles the setting up of the structures that describe the location of the sound sources, description of the room, and other minor details. Setup gets its information from Comm thread, and produces a structure that is sent to the Play routine for processing. This thread is involved in handling the detailed timing aspect of the sound projection. When the Setup decides it is time, the Play thread will process and project the information immediately.

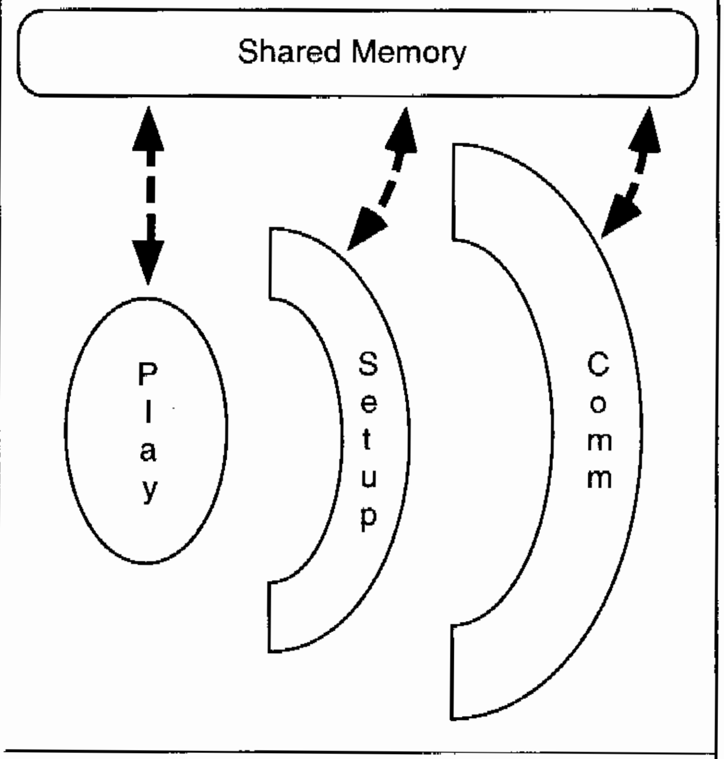

Figure 4-1: The three threads of SSSound. Play thread does all the processing, the Setup thread does all the timing and setting and the Comm thread holds higher-level programs such as a scripting language. All the threads share the same memory.

The Play thread is the most power-intensive and time-critical thread. Play takes the structures that are produced by Setup and puts them in a cycle of read / echo / reverb / play for each speaker. Each cycle processes manifold samples of sound, defined by the variable COMPUTE_SIZE . Currently COMPUTE_SIZE is set to 4410 samples(100ms), which is small enough to pass user input into the stream relatively quickly, but it is large enough not to be affected by the cycle overhead. Moreover the system is limited to two speakers per DEC Alpha computer (Alpha 3000/300 running under OSF 3.0), since the computation required for two speakers takes up most of the processing on the Alpha. The speaker limit is bound by two factors: first the computational limit; reverberation takes up two thirds of the processing for SSSound, while Echo takes up another 20%. Each speaker needs one reverberation routine, thereby cutting the limit of speakers to 2. Second the hardware limit; each LoFi card has only two outputs (left and right speakers) and it uses Alpha's Hi-speed Turbo channels, which are in high demand. The computation for the two speakers is conducted on separate pipelines (Figure 4-2). The processing for the first speaker is the more complete process, since all computation must be done from scratch, while the processing for the second speaker uses information already computed from the first.

4.1 The Play Thread

The Play Thread is responsible for the real-time computations and projections of the audio samples. It's processing pipeline is as described in Figure 4-2. The first element the program checks is the status of the system: has the state of the system changed in any way? The Play Thread is merely concerned with checking global variables it receives from the Setup Thread.

For the general case, let us assume that the state had changed. Play would get the new dimensions of the room, listener position/orientation, sound locations and/or other variables from the Setup thread. Using this information, Play calculates the filter taps for each speaker it is processing (described in more detail in Chapter 3). The filter taps are the end result of every state of the environment. Therefore, if the environment does not change, Play would skip to the process step of the cycle routine.

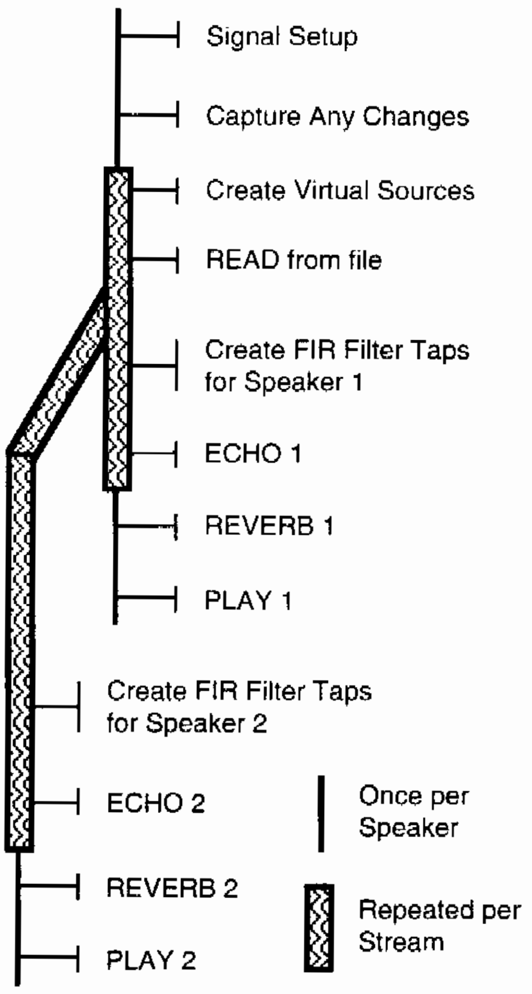

Figure 4-2: The PLAY process pipeline according to speakers and streams. Speakers 1 and 2 have different pipelines in order to conserve computational power.

Once the program has obtained the filter taps, either computed fresh from the new information or reused from the previous compute, then it is able to process the samples. Play would process every available stream by first reading COMPUTE_SIZE samples of the stream from the stored file into local memory. Then it would apply the echo filter taps calculated from the state of the system. The result of the echo is added into a play stream. After Play does this to every stream, the resultant stream would go through a reverberation process, which adds the appropriate reverberation to the stream depending on the characteristics of the simulated room. All these processes are described in the next subsections in more detail.

4.1.1 Read from file

Having calculated the filter taps for each speaker, the program is ready to read the samples from the file into our processing stream. The sound data files are stored as 16 bit integers in raw format (CD format). Every cycle, Play reads COMPUTE_SIZE integers into its read buffer (named read_buf) - which are COMPUTE_SIZE large. These integers can be accessed using two character reads, one offset by 8 bits. They then are converted to floats and stored in the

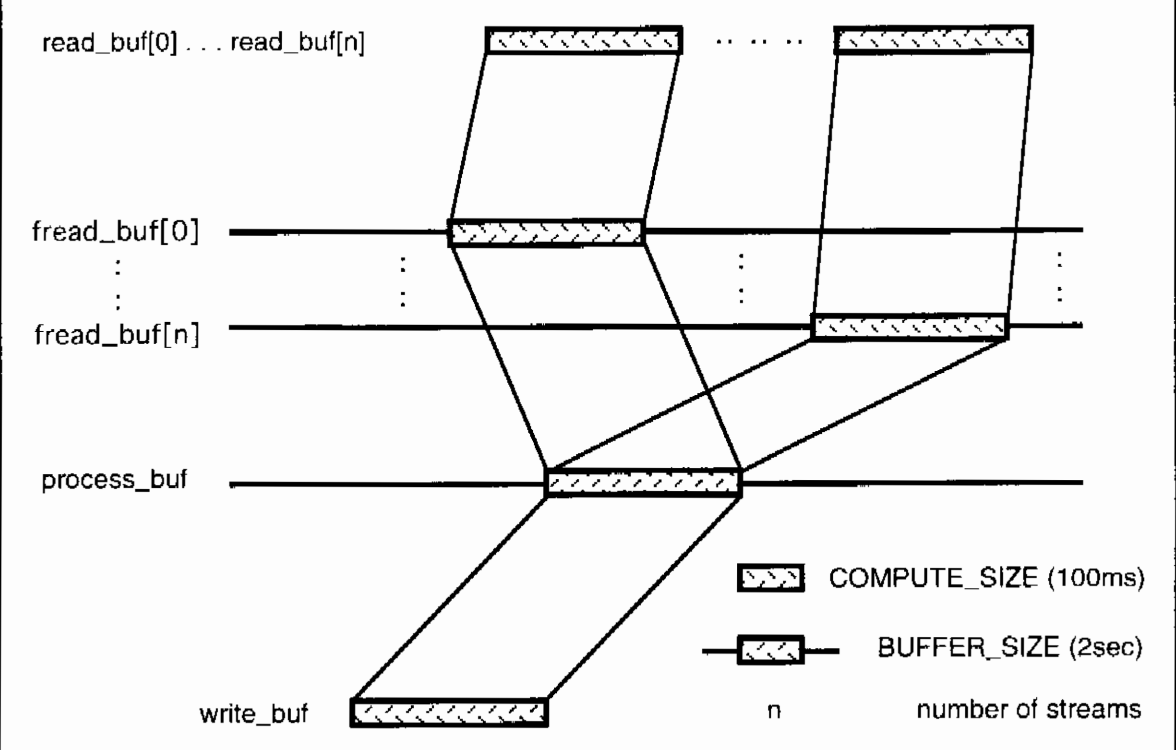

Figure 4-3: The Play Thread Buffer sizes and processing pipeline. First 16-bit integer samples are read into *read_buf[1...n]*. They are then converted to 32-bit floats and saved into the appropriate section in *fread_buf[1...n]*. The ECHO process is applied on the *fread_buf[1...n]* and the result added into the current section in *process_buf*. When all the streams have added their echoes result, REVERB is applied to the current section in *process_buf* and the result saved as 16-bit integers in *write_buf*. Those samples are then sent to the audio server to be played.

appropriate portion of the float read stream buffer (named fread_buf, length = BUFFER_SIZE). BUFFER_SIZE is currently set to 2.0 sec.

The rest of the processing is done in floats. Float to integer conversion is a computationally costly process, so it is not performed until the play cycle. 16-bit integer storage in character form is not feasible. Character access of 16-bit integers requires two memory accesses, a shift and an addition. The choice was left between storing the samples as 32-bit integers, or 32-bit floats. Floats were chosen as the method of storage for two reasons. First, because of the design of the Alpha chip, float calculations are on the average 20% faster compared to integer calculations [Alpha 92]. Second, float calculations allow the most significant digits to stay with the number, otherwise the system would be bound by certain limitations on integer calculations. This second reason also enables the program to have better estimates of the filters taps needed for processing. The Play thread is computationally very intensive; 112 shifts, 31 multiplies, 156 additions and 38 bitwise and operations are needed for every sample the program reads/plays. Therefore, any cycles saved will assist in program efficiency.

4.1.2 Echo

Once the samples are copied into the float read buffers, a process called Echo is applied. Echo adds the effect of sound reflecting off the simulated walls. This is an important consideration since all walls are somewhat reflective. The required information for the Echo process is :

- the specification of the room,
- the reflective coefficients of each wall,
- the maximum depth of reflective sources to be computed,
- the maximum number of reflections to be computed,
- the location of the sound source,
- the location of the listener,
- the orientation of the listener to the room,
- the location of the speakers.

The Echo process takes the above parameters and computes a set of filters for each speaker and for each stream. Each filter is essentially a set of delayed taps. Echo applies these filters over the read buffer for each stream. Here the particular implementation of the echo process is described.

The *room description* specifies the size and shape of the virtual room to be simulated. To reduce the computation, the room is restricted to being simply rectangular in shape. Thus only the length and the width of the room are needed (assuming the origin of the virtual world is the center of the room). The system is designed such that new types of room descriptions, such as polygonal rooms, can easily be implemented. Polygonal room descriptions are more effective since the do not restrict the shape of the room. However, it requires much more computation to actually find all the source reflections in the room

[Borish 84]. This thesis is more concerned with the general aspects of SSSound—the application of this technology to polygonal rooms is left for later studies.

The *reflective coefficient* of each wall is a number between 0 and 1 specifying the ratio of the reflected pressure to incident pressure. A coefficient of zero represents no reflection of sound. Each room has a general *maximum number of reflections* associated with it, and each stream has a *maximum depth* associated with it. These two parameters together add some control to the whole system so as to customize the acoustic environment more precisely.

The *location of the sound source* represents the point from which sound projection is simulated. It is described as a point in the z and x plane. In most cases, the sound location is the same as the location of the projected image in the virtual room. However, this could be overridden in the script so as to allow for sounds to appear from places where the image might not be, such as a call from behind the person, or from another room.

The *location of the listener* describes the position in the virtual room where the listener is, while the *orientation of the listener* describes what direction the listener is with respect to the room (the counter-clockwise angle from the minus-z axis). These two elements together correspond to the location and orientation of the camera/viewer in the visual domain.

The *location of the speakers* is the only dimension in the system that has any physical meaning. What this defines is the location of the speakers *in the real world* with respect to the front of the listener. The z direction is defined as the normal to the screen, therefore, if the listener is directly in front of the screen, he is looking in the minus-z direction. The speaker location is defined with respect to this axis system.

Sound locations in front of the speaker actually produce a negative tap delay—which in reality implies the need for a sound sample before the program has read it. However since the system does not read sources before they happen, a delay line was added to account for this phenomena. Echo sets a COMPUTE_SIZE delay to the pipeline—a tap at delay zero is actually a tap COMPUTE_SIZE away. Therefore the smallest tap delay the system can handle is minus COMPUTE_SIZE, which implies a source 3.42

meters in front of the speakers (if COMPUTE_SIZE is 100ms). This represents the maximum distance between listener and speakers for a full rendering of all the possible sound locations.

Echo receives COMPUTE_SIZE integer samples, and changes those to COMPUTE_SIZE float samples, saving them in a BUFFER_SIZE long buffer. The BUFFER_SIZE (2 sec) buffer is needed to ensure that echo sounds are stored to be used for echoed samples. For example, because of echoes, a sample might be needed that was actually projected 100ms before. This sample is stored in the fread_buf stream. Echo adds the above filtered stream into the reverb buffer (process_buf). Since this happens for every sound stream, the result of all the Echo routines is a single stream which contains all the sounds and their echoes from all the streams.

4.1.3 Reverberation

Reverberation is the general sound level that resides in a room from all the sound sources. No room is totally reverberation-free therefore the program needs to simulate some form of reverberation for a realistic effect. Most reverberators require large memory allocation and intensive computation, more than a general purpose processor can offer for real-time purposes. To circumvent this problem the program utilized nested all-pass filters (described in chapter 3).

Reverb receives the stream of the result of all the echoes as described above and a description of the room, and outputs a reverberated response to the sounds. The buffer used for reverberation is the same size as the Echo buffer, which also includes a 300ms extra buffer space at the beginning. The extra space is important to ensure continuous running of the system, without infinite memory. Whenever the program reaches the end of one buffer for echo or reverb, it copies the last 300ms of the buffer to the beginning and then returns the pointer to just after that 300ms. This ensures that not much data is lost in this transition. For the echo, 300ms for sound corresponds to about 100m, and most rooms are smaller than that. While for reverb, the biggest reverberator the system uses has a span of less than 300ms.

Reverb transforms the float echoes response to a single COMPUTE_SIZE play buffer of 16-bit integers. Float results of reverberation are never saved. As soon as a reverberated sample is computed it is typecast as integer and saved in the play buffer.

4.1.4 Play

The main function of the play cycle is to project the computed sound samples over the speakers. The server the program uses is the AudioFile(AF) server from Digital [Levergood 8/93]. AF handles all the play requests: it copies the samples into its own buffer, so the client can utilize its own buffers once the play function returns. AF also handles the play time requests. Requests for play at time before current time are ignored, while requests for play at time after current time are stored up. Thereby the play cycle is a simple cycle of informing AF where the information is, how long it is, and when to play it; the program then resumes its normal processing.

Timing is very important to the play cycle, as well as to the whole system. The system should not compute a sample after its intended play time has passed. If it takes longer for the computer to process the required information than it does for it to play the samples (i.e., 1.2 sec processing time for 1 sec sample length) it is necessary to discard the previously computed samples and jump ahead to real-time. Nor should the system compute samples too far ahead of current time, to ensure quicker interactive response. SSSound's answer to this problem is to compare the current process-time to the real-time at the beginning of every cycle. If process-time is ahead by MAX_AF_STRAY (set to 3*COMPUTE_SAMPLES = 300ms) then the program delays by calling the setup thread again, until real-time approaches process-time by at least 300ms. The author has found 300ms to be a reasonable time to ensure continuous play in a non-real-time system such as Unix. Since Unix operating systems run clean-up routines that override most routines, the 300ms lead provides a good buffer to fall back on when these routines are executed.

4.2 The Setup Thread

The setup thread is the thread which runs alongside the play thread. The Setup thread deals with all the synchronization and the communication to the external interface. In essence, Setup knows everything and tells Play what to do only when Play needs to know. Setup sees two sets of variables: *setup*_variables and *now*_variables (Figure 4-4). The *setup*_variables are assigned by an outside system, such as a scripting language, that calls the functions provided by SSSound (see section 5.1). The *now*_variables are set by the Setup thread and looked at by the Play thread.

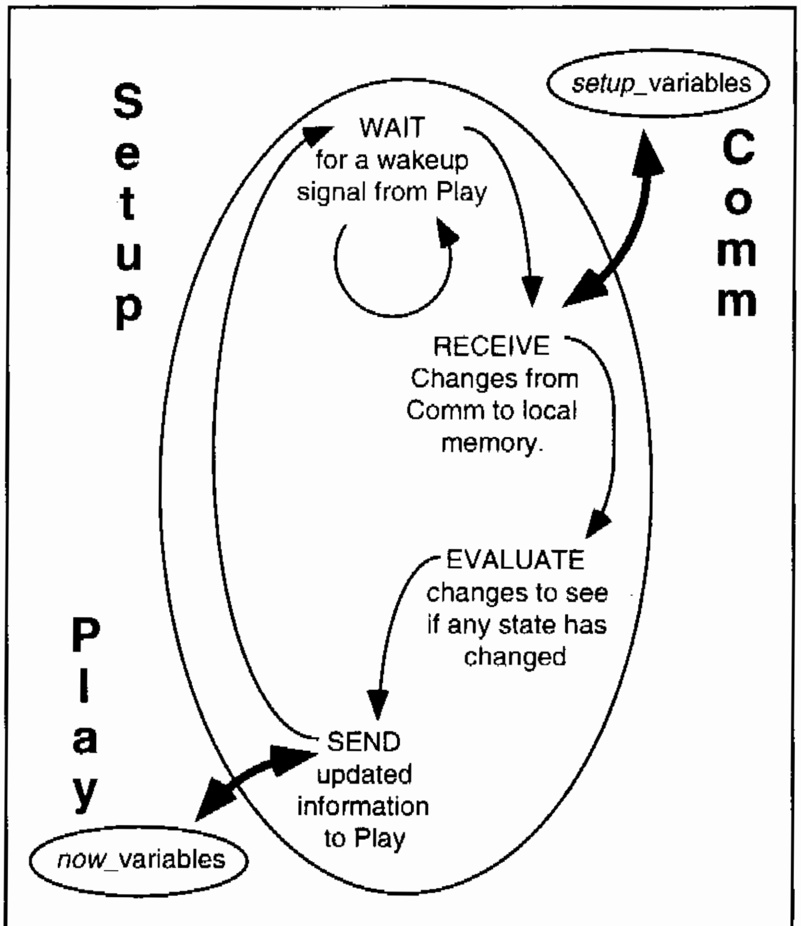
Setup
Comm
Play
setup_variables
now_variables
WAIT for a wakeup signal from Play
RECEIVE Changes from Comm to local memory.
EVALUATE changes to see if any state has changed
SEND updated information to Play

Figure 4-4: The SETUP process pipeline and its communication with the Comm thread using *setup*_variables and with the Play thread using *now_variables*.

Whenever the main thread calls an *sss*_function, that function changes the state of the *setup*_variables. The Setup function, once every 100ms at least, looks at the *setup*_variables and sees which ones have changed, and copies that information into its local-memory. Then Setup analyzes those user changes and formulates a system change if one is needed. For example, if a new sound should be played, then Setup checks if the time to play it has come. If so, it detects a system state change (i.e. from waiting to playing). Once a change is detected, Setup would capture the mutex between setup and play and write that change into the *now*_variables. Thus a complete setup cycle has been accomplished, and all changes are passed on to Play.

Setup then sleeps until Play signals the start of its new cycle. Then Setup is allowed another cycle through its Receive, Evaluate and Send processes. This eliminates any undue processing, as Play can only read the information every 100ms. Setup is also allowed to wake up whenever process-time is ahead of real-time by 300ms (as described in section 4.1.4).

One of the advantages of this multi-thread system is that the system has a thread which controls its setup, and another that performs all the processing. This allows Setup to control any synchronization needed to achieve real-time results.

4.3 Synchronization

Real-time synchronization for playback to users falls into two categories. The first is synchronization in a single stream, where each packet of information needs to be put out exactly after the one before it (serial synchronization). The second category is synchronization between two or more streams, such as an audio and a video stream. This type of system, where a packet from one stream needs to appear at the same time as a packet from another stream, is called parallel synchronization. SSSound's synchronization problem lies in the latter category.

SSSound in its most general form, as discussed in section 1.1, is part of a multi-media system. SSSound handles only the audio, while another platform, such as a system like Cheops [Bove 94], would handle the video. The distinction between a structured system such as SSSound, and a traditional or even interactive traditional system is that the structured system cannot predict what the next frame/sound will be. In a structured system, the video and audio frames are put together instantaneously, without prior knowledge of how they might be put together. In a traditional system, the frames are pre-computed and the only aspect of synchronization is to fetch and play the frames together. For a traditional interactive system, all frames are pre-computed, and at run-time, only the ordering of the frames changes. In this manner, a system could be set up to intelligently predict certain paths of interaction and pre-fetch the frames [Rubine 94]. In a structured system however, one can only pre-fetch original sound/image sources. The structured system takes these raw data and merges them together for every image frame or sound sample, according to certain rules defined in the scripting language.

A synchronization system proposed by Fabio Bastian and Patrick Lenders [Bastian 94], bridges the gap between the more traditional approach to videos, and the new structured approach. Bastian and Lenders assume their streams are completely independent of other events, and thus propose a formal method for specifying parallel synchronization: Synchronized Session. *"A synchronized session could be described as multiple independent, but related data streams, bound together by a synchronization mechanism that controls the order and time in which the information is presented to the user. " [Bastian 94]* In order to achieve this, they set up a synchronization file that requires certain events in the multiple streams to occur at the same time (Figure 4-5a). A system is set up to process the data and the synchronization files to achieve a synchronized output.

In a structured system, a similar setup to Bastian and Lenders is achieved by having the scripting language handle all the synchronization (Figure 4-5b). That is, the language knows when events should happen in both the audio and the video domain. The scripting language ensures that the instructions to process the video and the audio of a certain event are sent at the same time. Each of the audio and the video processors, then, attempts to process the information as quickly as it can. The information could be skewed if one of the streams processes the request faster than the other, although an allowance could be made in this event by adding certain delays in the particular pipelines of the scripting language. A somewhat reasonable result could be achieved with a system such as this, though not as good as certain synchronization techniques for more traditional video systems.

A constraint of this system is its inability to use other synchronization techniques because the two streams are never together except at the moment of conception. That is once the scripter gives instructions to process frames, it has no control over when the actual frame is created and delivered with respect to other types of frames. Both the audio and video processing units are on different platforms, and as a result, the two sections never meet except at the user. A system could be designed so that another synchronization process is added immediately preceding user reception using more tradition approaches [Ehley 94]. However that is beyond the scope of this thesis.

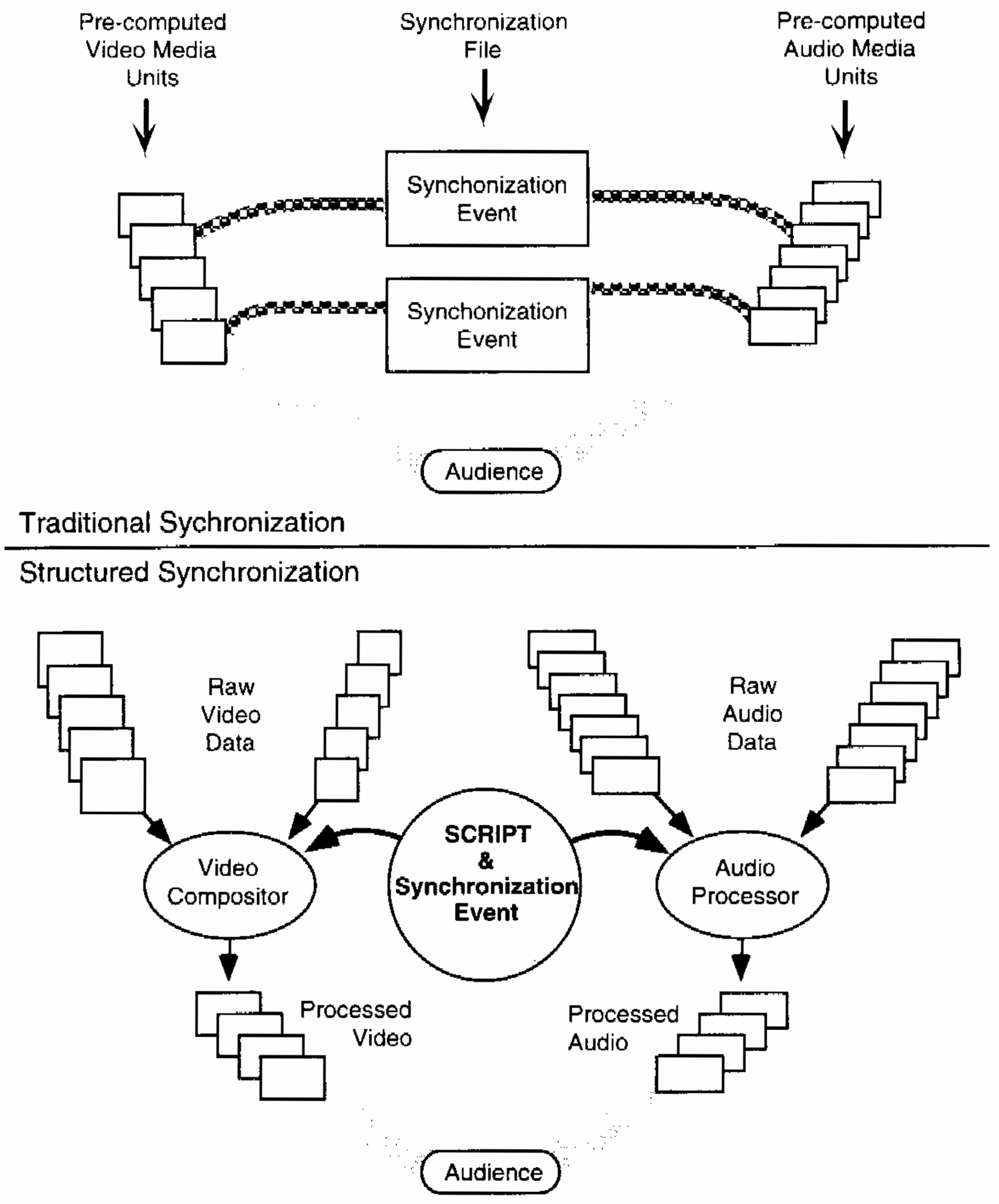

Figure 4-5: A comparison of the two types of synchronization processes. **a**) the Traditional System, like a Batian and Lenders system (top), has pre-computed frames and synchronization is only performed right before the audience views the results. **b**) the Structured System (bottom) computes the frames on the fly according to the script and user input. Synchronization is attempted by giving the Audio and Video processing instructions simultaneously.

5 External Interface

SSSound runs on a three thread system as discussed in chapter four. The third thread is the main thread, or Comm Thread (Figure 4-1). This thread is the thread responsible for running the section of code that gives instructions to the Setup thread using a pre-defined set of functions available to the user. The Comm Thread could either be a comprehensive scripting language or it can be an open socket which stays idle until it receives an instruction, whereupon it decodes it and runs the appropriate SSSound instruction as described in the following section.

5.1 External Functions

The origin of the instructions does not influence the action of the program; the instruction is simply carried out. The first instruction to be executed is the following:

- `int sss_sound_initialize(int speaker0, int speaker1);`

  This instruction sets up the two other threads, and decreases the Comm thread's priority to the lowest available so that the other threads can do all the time critical processing effectively. Once the two threads are created and all the buffers allocated, the two threads remain idle until they receive a starting signal from the Comm thread (`sss_start_play`). This allows the program to receive instructions off-line and to be ready to process them at the starting signal. This function also identifies for which speakers it is processing audio samples. This is useful if SSSound is running on more than one machine because it allows the simulation of more than two speakers. Each machine can then be given the same instructions, so long as the corresponding speaker numbers are supplied to `sss_sound_initialize`. The speaker numbers index a list of speaker locations.

Once the initialization routine is completed, the program is ready to accept any other instructions. However, the speaker locations need to be specified before the starting signal is sent.

- `int` **`sss_add_speaker`**`(float` *`xpos`*`, float` *`zpos`*`);`

  This instruction is to be executed for every physical speaker in the system. The speakers are to be added in clockwise order. The order in which they are added should correspond to the speaker number specified by `sss_sound_initialize`, such that the first speaker added is number 0, the second number 1, and so forth. The positions are in meters, and are relative to the listener, where the plus-x is to the right of the listener, and minus-z is to the front of the listener.

  NOTE: If only two speakers are added into the system, a third one is automatically created on the other side of the listener equidistant from the two physical speakers.

Once the speaker locations are defined, the program can start. However, it is recommended that all sounds be registered before the starting, and that some form of room description, listener position, and other general characteristics be supplied.

- `int` **`sss_start_play`**`();`

  This instruction starts the loop in the play-echo-reverb-play pipeline. Before this can be executed, *all* speakers need to be added into the system.

Once `sss_start_play` is called, any of the following functions can be called in almost any order.

- `int` **`sss_register_sound`**`(char *` *`filename`*`);`

  This instruction registers a *`filename`* and returns a *`sound_id`*. Whenever the filename is needed, the *`sound_id`* is used to identify the filename so as to reduce the non-critical information passing between the threads. Thus it is recommended that all the registering of sound be done before the sss_start_play.

  RETURN: *`sound_id`* representing the filename.

- `int` **`sss_play_sound`**`(int` *`sound_id`*`, float` *`xpos`*`, float` *`ypos`*`, float` *`zpos`*`, float` *`start_time`*`, float` *`stop_time`*`, float` *`max_depth`*`, float` *`volume_db`*`, int` *`play_mode`*`);`

This instruction requests that a specified sound be played. It takes in a `sound_id` from the registration process, and the location (`xpos, ypos, zpos` in meters) of the sound source. It also takes `max_depth` of the sound, which represents the maximum allowed depth from the listener to any of the virtual sources, and a `volume_db` which is the relative volume in dB of the source. The timing of the sounds in the system comes in many forms (the capitalized text are modes to be 'or'ed with `play_mode`).

There are two ways to start the sound :

- now: SSS_START_NOW [no `start_time` needed]
- at `start_time` : SSS_START_TIME

There are three ways to end the sound:

- at file end: SSS_END_FILEEND [no `end_time` needed]
- at `stop_time`: SSS_END_TIME
- at `stop_time` with looping: SSS_END_TIME_LOOP

The start mode and end mode are 'or'ed together to make up the `play_mode`.

e.g., SSS_START_END_DEFAULT mode is SSS_START_NOW | SSS_END_FILEEND

There are two modes of projecting sound. The normal mode is to localize the sound source as described in this thesis. The other mode is surround mode, where the same sound source comes from every speaker at the same volume. This is the SSS_PLAY_CIRCULAR mode which can be 'or'ed in with the `play_mode` as above. This mode ignores the location information, and only uses the `volume_db` variable to define how loud this sound source should be from every speaker.

RETURN: `play_sound_id`, which is a unique id for this sound play request.

NOTE: Before this function is called, the room descriptors, the listener position and the speaker position need to have been defined.

- `int` **`sss_change_sound`**`(int play_sound_id, int change_mask, float xpos, float ypos, float zpos, float start_time, float stop_time, float max_depth, float volume_db, int play_mode);`

This function changes the current sound description by taking a `play_sound_id`, which is the sound id returned from sss_play_sound request, and `change_mask` which is an 'or'ed number of the following:

SSS_CHANGE_XPOS
SSS_CHANGE_YPOS
SSS_CHANGE_ZPOS
SSS_CHANGE_START_TIME
SSS_CHANGE_STOP_TIME
SSS_CHANGE_DEPTH
SSS_CHANGE_VOLUME
SSS_CHANGE_MODE

The rest of the inputs are as described in `sss_play_sound`.

RETURN: 1 if successful, 0 if not.

- `int` **`sss_delete_sound`**`(int` *`play_sound_id`*`);`

This instruction takes a `play_sound_id` and stops it. If for some reason the sound has already stopped (if it reached the end of the file in a SSS_END_FILEEND mode), this function does nothing.

RETURN: 1 if successful, 0 if not.

- `int` **`sss_set_room_desc`**`(float` *`x_wide`*`, float` *`y_wide`*`, float` *`z_wide`*`, int` *`max_reflect`*`, float` *`ref_coef_0`*`, float` *`ref_coef_1`*`, float` *`ref_coef_2`*`, float` *`ref_coef_3`*`,float` *`ref_coef_top`*`, float` *`ref_coef_bottom`*`);`

This function sets the room description dynamically. `x_wide, y_wide, z_wide` is the length of the room in meters in the x-, y-, z- directions. It is assumed that the origin is the center of the room. `max_reflect` is the maximum number of reflections/echoes to consider in calculating the virtual sources. `ref_coef_[0,1,2,3]` are the *pressure* reflective coefficients of the vertical walls in the room in clockwise order starting from the minus-z axis (which is the wall facing the listener). `ref_coef_[top, bottom]` are the pressure reflective coefficients of the ceiling and floor, respectively. (`ref_coef[..]` is a number between 0 and 1, where 0 is *no* reflection).

RETURN: 1 if successful, 0 if not.

NOTE: Only horizontal information is used for actual position calculations. The vertical height (`y_wide`) and the `ref_coef_[top, bottom]` are used only to calculate the acoustic quality of the room.

- `int` **`sss_set_other`**`(float` *`room_expand`*`, float` *`sound_expand`*`, float` *`reverb_gain`*`);`

This instruction sets the variables as follows:

- *`room_expand`* is a coefficient that expands the seeming size of the room with respect to the center of the room,
- *`sound_expand`* is a coefficient that expands the seeming location of the sound source with respect to the listener's position.
- *`reverb_gain`* is a coefficient that defines how much reverb to have.

NOTE: For any *`reverb_gain`* lower than MIN_GAIN_FOR_REVERB (set to .1 in sss.h) there will be *no* reverb applied to the stream.

RETURN: 1 if successful, 0 if not.

- `int` **`sss_set_listner`**`(float` *`xpos`*`, float` *`zpos`*`, float` *`angle`*`);`

This function sets the listener location (*`xpos, zpos`*) in the virtual room with respect to the center of the room. It also sets the *`angle`* which is the direction the listener is looking, taken counterclockwise from the minus-z axis in radians from -pi to +pi.

RETURN: 1 if successful, 0 if not.

- `float` **`sss_get_time`**`();`

This function returns the current processing time (in seconds) of the system.

RETURN: time of SSSound system (in seconds).

- `int` **`sss_set_time`**`(float` *`time`*`);`

This sets the current processing time to *`time`* in seconds.

RETURN: 1 if successful, 0 if not.

- `void sss_set_verbose(int verbose);`

This function sets the verbosity of SSSound as follows:

level -1:	only timing information.
level 0:	no verbosity.
level 1:	main instructions but without sss_change_sound calls,
level 2:	add sss_change_sound calls
level 3:	add timing information, buffer ends,
level 4:	add yield calls

- `void sss_yield_to_sound();`

This function is a command which allows the Comm Thread to yield its hold over the CPU to the sound thread. This is needed when it is apparent that the Comm thread is taking up too much of the processing power, and thereby hindering the time-critical processing of the Play and Setup threads. One way to definitely test the Comm thread load is to run a profiler over this program; if the programs in Comm thread are using more than 20% of the CPU cycles (normal numbers are closer to 10% or less), then the Comm thread is using too much power. When this call is executed the Comm thread gives up its hold over the processor to the other two threads, so that the other threads can do their time-critical jobs. Assuming that it requires much less computation to calculate the state of the system than to actually process it, the Comm thread should not compute states faster than the decoder can process them.

NOTE: when the program runs `sss_sound_initialize`, the priority of the Comm thread is reduced to minimum, while the priority of the Setup and Play threads are increased to maximum.

5.2 An Example: Isis

Using the above instructions, one can dynamically change the state of any of the sounds, as well as the room quality and other characteristics. SSSound is not intended as a stand-alone system. It is intended to be hooked onto a higher level program that can handle the sound as another aspect of a structured object. The higher level program could be a script interpreter.

The current system in the Media Laboratory uses a script language named Isis [Agamanolis 96]. This scripting language was modified to accept and send audio instructions. In just a few lines, one can describe a room and design a system where any of the characteristics of the room or sound can change.

Below is an example of an Isis script that sets up a room and moves one sound source left and right in cycles. The lighter and smaller font is the actual script, the bolder font is the explanation.

```
# ----------------------------------------------------------------------
# This is an audio test script.

(load "strvid.isis")
```

Loads a script which creates special data types for the system.

```
# ----------------------------------------------------------------------
# Initialize sound with speaker positions

(initialize-audio)
```

Creates two other threads, and waits for setup information and for the starting signal. (≡ `sss_sound_initialize`)

```
(set-sound-verbose 0)
```

Sets the verbosity level to no verbosity. (≡ `sss_set_verbose`)

```
# ----------------------------------------------------------------------
# Register sound files

(set soultattoo
     (register "/cheops/araz/sound_file/soultattoo.raw"
             "Sound 1" ft-raw-sound))
```

Registers the sound and sets the variable *soultattoo* to an identification number for that sound. (≡ `sss_register_sound`)

```
# ----------------------------------------------------------------------
# Set speaker positions

(set-speakers (Poslist (Pos -1.0 0.0 -1.0)
                       (Pos 1.0 0.0 -1.0)))
```

Calls sss_add_speaker twice with two speaker positions in clockwise order. (≡ `sss_add_speaker`)

```
# ----------------------------------------------------------------------
# Create structures
```

These next few instructions create Isis internal structures.

```
(set movie (new-internal-movie))
(set scene (new-internal-scene))
(set stage (new-internal-stage))
(set view (new-internal-view))
(set disp (new-internal-display))
(set sound1 (new-internal-sound))

(update movie
      mov-name "The Sound Tester Movie"
      mov-scene scene
      mov-display disp)
```

A *movie* is composed of a *scene* and a *display*.

```
(update scene
      sc-name "The only scene in the movie"
      sc-view view
      sc-stage stage
      sc-sounds (Addrlist sound1))
```

A *scene* is built up of a *view*, a *stage*, *sounds* and *actors*.

```
(update stage
      st-name "The wacko stage"
      st-size (Dim 5.0 3.0 5.0)
      st-reflection-coefs (Coeflist 0.5 0.7 0.5 0.7 0.5 0.5)
      st-max-reflections 10
      st-size-scale 1.0
      st-dist-scale 1.0
      st-reverb-scale 0.0)
```

The *stage* represents a certain description of a room. The author can create many *stages*, and can switch from one *stage* (with its varying acoustic qualities) to another, by simply entering a different *stage* name. (≡ `sss_set_room_desc`)

```
(update view
      vi-ref-point (Pos 0.0 0.0 0.0)
      vi-normal (Pos 0.0 0.0 1.0)
      vi-up (Pos 0.0 1.0 0.0)
      vi-eyedist 1.0)
```

Represents the computer graphics method of expressing where the user is looking. The *normal* is a vector that points towards the viewer.

```
# ------------------------------------------------------------------------
# Set up to play some sounds

(update sound1
      snd-sound-object soultattoo
      snd-start-time -1.0
      snd-end-time 10000.0
      snd-position (Pos -1.0 1.0 -1.0)
      snd-volume 0.0
      snd-max-depth 100.0
      snd-loop True)
```

Sets up an internal sound structure to play a sound with certain characteristics. (≡ `sss_play_sound`)

NOTE: start-time = -1.0 in Isis means start *now*.

```
(set pos1 (new-timeline (Pos 0.0 0.0 -1.0)))
```

Creates a new timeline, which is a special data structure for specifying time-varying quantities.

```
(key pos1 0 (Pos -3.0 0.0 -1.0))
(key pos1 10 (Pos 3.0 0.0 -1.0) linear)
(key pos1 20 (Pos -3.0 0.0 -1.0) linear)
```

Defines positions in time, sets their locations and asks the interpreter to interpolate linearly between those points. In this case, the position at time 0s is 3 meters to the left and 1 meter in front. Then at time 10s, the position reaches 3 meters to the right, and then 3 meters to the left again at time 20s.

```
# ------------------------------------------------------------------------
# Start the sound server

(start-audio)
```

Sends instruction to begin sound processing. (≡ `sss_start_play`)

```
(set time 0.0)
```

Sets local time to 0.0sec.

```
(while True
```

Loops through the following without stopping.

```
(begin
     (set time (get-sound-time))
```

Gets the SSSound time and makes the variable `time` equal to it. (≡ *sss_get_time*)

```
     (update sound1
           snd-position (pos1 time))
```

Updates the internal structure with its new position for that point in time. (≡ *sss_change_sound*)

```
     (build-frame movie)))
```

Sends all the changes made during this time loop to the sound server.

6 Thoughts and Conclusions

The audio rendering system described in this thesis is implemented at the MIT Media Laboratory. The audio system operates in real-time with four sound source localizations and reverberations for each of the two speakers at a rate of 44.1 KHz. The locations of the sources, the room characteristics and the listener perspective can be dynamically changed based upon user input. SSSound runs with a scripting language, called Isis, developed for interactive environments. All the structure for the synchronization described in this thesis is implemented. At the time of this thesis deadline however, the scripting language was not in a state that made it feasible to test the synchronization.

Overall the system runs smoothly. There are some problems with this system - mainly dealing with the non real-time operating environment. The user needs to be careful, when running the audio rendered, with his interaction with the operating system (UNIX). Any action performed by the user that causes UNIX to stop its operation for more than 300ms in response, causes a momentary audio interruption.

The current system delay includes 100 milli-seconds for the objects that should appear in front of the speaker (described in section 4.1.2). The delay also incurs a 300ms buffering delay which is not apparent to the user if instructions are pre-computed; however it is apparent in interactive instructions. Moreover in interactive systems, since instructions are only processed every 100ms, there is also a delay associated with the time the instruction is given in the 100ms cycles. Therefore for non-interactive instructions, the delay is 100ms(for the objects in front of the speaker), while for interactive instructions the delay is 400-500ms. In reality not many users of the system noticed the delay.

The system could undergo improvements in a couple of areas. One improvement is to have more than two speakers to increase spatial localization cues. This could be achieved in one of the following ways:

- The system could be processed at half the sampling rate of 44.1KHz in order to have enough processing power to process data for four speakers. Because of hardware limitations[1] though the system would still have to use a LoFi card on another machine. Therefore a connection would need to be opened to this other machine and sound samples need to be sent there.
- The system could have two Alpha workstations connected to each other both running SSSound, but with only one running the scripting language. And a socket could be opened between the scripting language program and the SSSound program on the other machine. With this configuration, every time the language gives an instruction to the first render, it sends an identical instruction to the render on the other machine (as long as the two renders were initialized with the right speaker values as described in section 5.1).

Another area worth examining would be the use of other methods of performing reverberation or localization of sound. This system setup uses an Alpha workstation's full power to achieve a real-time rendering for two speakers at 44.1 KHz. It could be possible with advances in technology and technique to process more than two speakers. The field of real-time localization is relatively new, and therefore there is potential for better results with more research.

Synchronization is another aspect worth examining. The question of synchronization has been studied and discussed extensively; however, not much work has been done in the area of interactive synchronization. The limitation lies in the interactive system's inability to predict the next audio or video frame. It is possible, though, to implement a synchronization step right before user viewing. However, this might increase the interactive delay of the system because the system would need to buffer to ensure that the audio or video frames are synchronized.

[1] All the Alpha workstations in the Laboratory are equiped with three Turbo slots. One is used for the video display, one is used for network connection, and only one is left to be used for audio. There are Alpha workstations equiped with more than three Turbo slots; however, they are beyond the budget constraints of this project.

7 Bibliography

Stefan **Agamanolis**, *High-level scripting environments for interactive multi-media systems*, SM. Thesis in writing, MIT, February 1996

Alpha, *Alpha Architecture Handbook,* Digital Equipment Corporation, Maynard Ma. 1992

Fabio **Bastian** and Patrick Lenders, *Media Synchronization on Distributed Multimedia Systems*, Proc. IEEE International conference on Multi-Media Computing and Systems, May 1994 Pg. 526

Jeffery **Borish**, "Extension of the Image Model to Arbitrary Polyhedra," *J. Acoustical Society of America.* 75 (6), 1984.

V. Michael **Bove**, Jr., Hardware and Software Implications of Representing Scenes as Data, Proc. International Conference on Acoustics, Speech, and Signal Processing 1993

V. Michael **Bove**, Jr., Brett Granger and John A. Watlington, *Real-Time Decoding and Display of Structured Video*, Proc. IEEE International conference on Multi-Media Computing and Systems, May 1994 Pg. 456

V. Michael **Bove**, Jr., and John A. Watlington, *Cheops: A Reconfigurable Data-Flow System for Video Processing*, IEEE Transactions on Circuits and Systems for Video Technology. VOL. 5, NO. 2 April 1995

Dolby Surround Sound, *Dolby Surround Sound Past, Present, Future,* http://www.dolby.com/ht/ds&pl/ppf-cont.html

Lynnae **Ehley**, Borko Furht and Mohammad Ilyas, *Evaluating of Multimedia Synchronization Techniques,* Proc. IEEE International conference on Multi-Media Computing and Systems, May 1994 Pg. 514

William G. **Gardner**, *The Virtual Acoustic Room*, SM. Thesis, Massachusetts Institute of Technology, September 1992

William G. **Gardner**, *Reverberation Algorithms*, in M. Kahrs and K. Brandenburg (Eds.), *Applications of Signal Processing to Audio and Acoustics*, Kluwer Academic Press 1995 (in press)

Brett **Granger**, *Real-Time Structured Video Decoding and Display*, SM. Thesis, MIT, February 1995

T. M. **Levergood**, A. C. Payne, J. Getts, G. W. Treese and L.C. Stewart, *AudioFile: A Network-Transparent System for Distributed Audio Applications*, CRL Technical Report 93/8, Digital Equipment Corporation, Cambridge Research Lab, 8/1993

Thomas M. **Levergood**, *LoFi: A TURBOchannel audio module.* CRL Technical Report 93/9, Digital Equipment Corporation, Cambridge Research Lab, 9/1993

James A. **Moorer**, *About This Reverberation Business,* Computer Music Journal, Vol. 3, No 2, 1979

Dean **Rubine**, Roger Dannenberg, David Anderson and Tom Neuendorffer, *Low-Latency Interaction through Choice-Points, Buffering, and Cuts in Tactus*, Proc. IEEE International conference on Multi-Media Computing and Systems, May 1994 Pg. 224

Irene **Shen**, *Real-Time Resource Management for Cheops: A Configurable, Multi-Tasking Image Processing System*, SM. Thesis, Massachusetts Institute of Technology, September 1992

G. **Theile**, and G. Plenge, *Localization of Lateral Phantom Sources*, J. Audio Engineering Society, Vol. 25, No. 4, 1977.

Barry **Vercoe**, and Miller Puckette, *Synthetic Spaces - Artificial Acoustic Ambiance from Active Boundary Computation*, unpublished NSF proposal, 1985. Available from Music and Cognition office at MIT Media Lab.

E. M. **Wenzel**, *Localization in virtual Acoustic Displays, Presence: Teleoperators and Virtual Environments*, 1992, 80-107