Université d'Ottawa • University of Ottawa

PERMISSION DE REPRODUIRE ET DE DISTRIBUER LA THÈSE

PERMISSION TO REPRODUCE AND DISTRIBUTE THE THESIS

NOM DE L'AUTEUR / NAME OF AUTHOR:	SENKI, Adel
ADRESSE POSTALE / MAILING ADDRESS:	53-135 Des Jonquilles Hull, Québec J9A 2L4
GRADE / DEGREE:	**ANNÉE D'OBTENTION / YEAR GRANTED**
M.Sc.(Systems Science)	2000
TITRE DE LA THÈSE / TITLE OF THESIS:	
DEVELOPMENT OF A WINDOWS™ 3-D SOUND SYSTEM USING BINAURAL TECHNOLOGY	

L'auteur permet, par la présente, la consultation et le prêt de cette thèse en conformité avec les règlements établis par le bibliothécaire en chef de l'Université d'Ottawa. L'auteur autorise aussi l'Université d'Ottawa, ses successeurs et cessionnaires, à reproduire cet exemplaire par photograhie ou photocopie pour fins de prêt ou de vente au prix coûtant aux bibliothèques ou aux chercheurs qui en feront la demande.

Les droits de publication par tout autre moyen et pour vente au public demeureront la propriété de l'auteur de la thèse sous réserve des règlements de l'Université d'Ottawa en matière de publication de thèses.

The author hereby permits the consultation and the lending of this thesis pursuant to the regulations established by the Chief Librarian of the University of Ottawa. The author also authorizes the University of Ottawa, its successors and assignees, to make reproductions of this copy by photographic means or by photocopying and to lend or sell such reproductions at cost to libraries and to scholars requesting them.

The right to publish the thesis by other means and to sell it to the public is reserved to the author, subject to the regulations of the University of Ottawa governing the publication of theses.

N.B. LE MASCULIN COMPREND ÉGALEMENT LE FÉMININ

Feb 06/01
DATE

Adel Senki
(AUTEUR) SIGNATURE (AUTHOR)

Université d'Ottawa · University of Ottawa

Development of a Windows™ 3-D Sound System using Binaural Technology

By
Adel Senki, B. A. Sc.

A thesis submitted to the Faculty of Graduate and Postdoctoral Studies in partial fulfillment of the requirements for the degree of Master of Science in Systems Science.

Faculty of Administration
University of Ottawa

June 30, 2000

Abstract

It is anticipated that 3-D sound will become a major component in virtual reality environments. In this thesis, we investigate the development of a Windows ™ 3-D sound system using binaural technology. This work should eventually be integrated in complex virtual environment software. The 3-D sound effects are achieved by applying a filter pair representing audio cues from a specific point in space to the left and right channels of headphones. As a first step, an offline 3-D audio file generation system is developed, and effects such as sound reflections in a virtual rectangular room are added to this system. The real-time implementation is considered next. The first approach tested for the real-time implementation uses the time domain convolution of a sound source signal with head-related transfer functions, but the computational load of this approach is such that it cannot run in real-time at audio sampling rates. As an alternative, a second approach with a Fast Fourier Transform overlap and save technique is used to reduce the computational load, and a real-time implementation at audio frequencies is therefore successfully achieved.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Dr. Martin Bouchard. Thank you for giving me the opportunity to participate in a 3-D sound reproduction system research project. I really enjoyed working under your guidance, leadership and supervision.

Many thanks to Dr. Tet Yeap and Dr. Jean-Michel Thizy Director of the Systems Science program, and to Dr. Bouchard for procuring me with more than adequate research facilities.

Special thanks to Dr. Yeap for an arrangement with Dr. Martin Bouchard to be my official supervisor, until Dr. Martin Bouchard got accepted in the Faculty of Graduate and Postdoctoral Studies.

I would also like to thank my wife, Amel El-hadi for providing me with full support and love throughout my studies. Many thanks to some of my working colleagues, especially Mr. Yu Feng and Mr. Mustafa Elarbi. Thank you all for your comments, help, and exchange of knowledge and friendliness throughout the course of my studies. It was really appreciated.

Table of contents

List of Acronyms

HRTF	Head-related Transfer Functions.
HRIR	Head-related Impulse Response
ITD	Interaural Time Differences.
IID	Interaural Intensity Differences.
DLL	Dynamically Linked Library.
A/D	Analog to Digital.
D/A	Digital to Analog.
FFT	Fast Fourier Transform.
DSP	Digital Signal Processing.
PCM	Pulse-Code Modulation.
API	Application Programming Interface

List of Tables

List of Figures

Chapter 1 Introduction

1.1 Motivation

In 3-D sound reproduction (or 3-D audio), signal processing techniques are used to artificially reproduce sounds that are perceived by human listeners as if they were "natural" sounds, although the sounds are actually reproduced via headphones or loudspeakers. To feel that the sound is "natural", listeners must have an appropriate perception of the location of the source (distance, azimuth, and elevation) and also a perception of the environment around them (open or closed room, small or large room, etc.) [1,2].

Binaural technology [3] is a particular technique for generating 3-D audio, where the transfer functions between a sound source and a human listener (called the head-related transfer functions, or HRTFs) are used to filter mono sound sources. Binaural hearing works on the basis that since we only have two ears, any sound we can hear can be reproduced by a pair of loudspeakers or by headphones. When sound waves arrive at the head, they are modified acoustically in a directionally dependent manner by diffractive effects around the head and by interaction with the outer ears. This is characterized by the HRTFs. Filtering a mono sound source with these HRTFs and playing back the resulting signals in headphones, the result is sounds that are perceived as originating from the source position specified by the HRTFs. When loudspeakers are used with binaural techniques instead of headphones, the resulting techniques are called transaural technique

[4]. There are also geometric approaches such as ambisonics and ambiophonics [5,6] to reproduce 3-D sound in a room, but those approaches usually require many loudspeakers, while the binaural or transaural techniques require only two loudspeakers for one listener. The drawback of binaural or transaural techniques, however, is that they typically have a high computational load.

There are many applications of binaural 3-D audio. There is of course the reproduction of sounds as they would be perceived in a known environment. For example, a listener hears sounds as they would be perceived in a great concert hall, although the listener may in fact be in a small office, with poor acoustical characteristics. There is also the artistic creation of sounds that are not common in real-life. It is anticipated that there will be a huge growth in the development of virtual reality environments (the most important application in the short term is for the entertainment industry, in particular for video games), and 3-D sound will be a significant component in such environments. This is the main motivation of this work.

1.2 Objectives and research methodology

The first objective of this thesis is to review and describe the basic sound spatialization theory required to understand how it is possible to modify mono sound recordings using binaural technology so that 3-D sounds can be perceived by a listener. The second objective of this thesis is to implement a real-time 3-D sound system based on head-related transfer functions (HRTFs), to be used in a Windows ™ software environment.

The sound system will work in real-time, and it will use a standard PC audio board. The second objective will answer questions such as:

- Is an average PC using Windows 98 ™ (an operating system not particularly known for its real-time capabilities) sufficient to produce 3-D sounds in real-time using binaural technology?
- Is a time domain approach (i.e. basic convolution) appropriate to implement the HRTF filtering?

In summary, the research methodology to be followed will be:

1. Review the sound spatialization theory

There are many aspects to the sound spatialization theory. It is important to identify which aspects (i.e. which cues) of sound spatialization can be reproduced in a real-time 3-D sound system based on binaural technology and head-related transfer functions. Only when this is clearly understood can a designer really know what he can expect in terms of 3-D effects performance from the 3-D sound software to be developed. This is why Chapter 2 is a very important chapter of this thesis.

2. Develop an offline 3-D sound system

This step requires finding the time-domain equations needed to compute the left and right channel 3-D sound signals from an input mono sound source, using the head-related transfer functions. It was chosen to de-couple the programming of the 3-D sound algorithms (i.e. this step in the methodology) and the programming of the real-time

acquisition and rendering (i.e. the next steps). This is because both of these tasks are challenging, and implementing them simultaneously in parallel seemed like an ambitious (or dangerous) approach. Therefore a version of the 3-D sound system was developed for an offline system that reads an input mono sound file and writes a stereo 3-D sound file, to be heard via headphones. This version does not have any real-time constraint, and the focus can therefore be put on the programming and debugging of the 3-D sound algorithm.

3. Develop a real-time version of the 3-D sound system

This step requires to modify the 3-D sound system from the previous step in order to write in real-time stereo output samples to a soundboard (again to be played via headphones). This requires using multi-thread programming combined with the use of Windows™ multimedia libraries. Note that in our implementation the input source is still a file, and not a microphone or a packet stream received from a computer network.

4. Improvement of the performance of the real-time system

From a computational point of view, it is possible to improve the performance of the real-time system. The net practical result will be to increase the maximum sampling rate of the audio sound signals to be reproduced by the system. This is critical since the HRTFs functions that are used in this thesis were sampled at audio frequencies. This step involves the use of frequency domain convolution techniques, in order to reduce the computational load of the 3-D sound algorithm. More specifically, an overlap and save fast Fourier transform (FFT) technique will be implemented. This will add to the

programming complexity of the program, but it will result in a more efficient implementation of the 3-D sound system.

1.3 Thesis Outline

The rest of the thesis is structured according to the following outline: Chapter 2 presents an qualitative overview of the theory of sound spatialization. Important topics are discussed such as: physical variables, psychophysical variables, coordinate systems, azimuth cues, interaural time differences (ITDs), interaural intensity differences (IIDs), elevation range cues, propagation effects and reflections.

Chapter 3 explains how an offline 3-D sound system that uses HRTFs and generates 3-D audio output files was first developed. In that part of the project it was possible to include environmental effects such as reflections and distance cues, because there were no real-time constraints.

Chapter 4 discusses the design of the real time 3-D sound system. The HFTF filtering was first implemented in the time domain using basic linear convolutions, and the performance of the system was evaluated. Chapter 4 then describes how the performance of the real time 3-D sound system can be improved using frequency domain methods to perform linear convolutions. The thesis concludes with Chapter 5, which summarizes the main results of this thesis and provides suggestions for future research activities.

1.4 Research Contributions

The main contributions of this thesis are:

- To briefly qualitatively summarize the theory of sound spatialization required understanding the concept of producing 3-D sounds using binaural technology.
- The development of real-time Windows ™ software for 3-D audio rendering using binaural technology. An off-line version (i.e. file based) of the software was also developed, and it was possible in this case to add some environmental effects such as reflections and distance cues.

Chapter 2 A brief overview of sound spatialization theory

This chapter reviews some basic aspects of sound and sound spatialization. It is very important that a programmer be aware of these aspects, in order to know which aspects (i.e. which cues) of sound spatialization can be reproduced in a real-time 3-D sound system based on binaural technology and head-related transfer functions. For example, it will be explained in this chapter that HRTFs can provide the cues for the position of a sound source, but unless they are combined with some other audio processing, HRTFs do not provide information about the acoustical environment (dimension of space, type of walls, presence of objects, etc.).

2.1 Acoustics and psychoacoustics

Sound spatialization is related to both acoustics and psychoacoustics. Psychoacoustics is a discipline that studies the relation between sounds (acoustics) and our perception (psycho). It is usually considered to be within the field of psychology. Acoustics is known as the science of sound. Or, in technical terms "... the generation, transmission and reception of energy in the form of vibration waves in matter." [7]. A huge number of scientific problems and disciplines fit into this definition, e.g. noise control, vibration and structural acoustics, electro-acoustics, room acoustics, building acoustics, musical acoustics, psychoacoustics, etc.

Acoustics is by its very nature a very inter-disciplinary field, drawing people from widely differing backgrounds. A person who works as an 'acoustician' might be a physicist studying acoustic wave propagation, a mechanical engineer trying to control noise and vibration, an electrical engineer designing a new electroacoustic transducer, a civil engineer designing the acoustic properties of a building, an experimental psychologist studying psychoacoustics, a physician doing research in audiology, a computer programmer designing the sound effects for the newest computer game, etc.

2.2 A simple example of sound localization

Sound originates from a disturbance of the air by any object [8]. For example, two hands clapping cause a disturbance of the air around the hands, and the hands are the source of the sound. The local region of air has increased energy caused by the motion of the air molecules. This energy spreads outwards in sound waves. For example, if a stone is dropped in the middle of a lake, this creates concentric waves. The waves travel out from the center as shown in Figure 2.1. An observer at any point on the lake would notice that the first wave to pass would be the largest in amplitude. Each subsequent wave would be a little smaller, until the lake is again calm. Note that an observer looking from the sky could not determine from which direction the stone came in the lake. This is the same effect as sound waves hitting a flat surface [9].

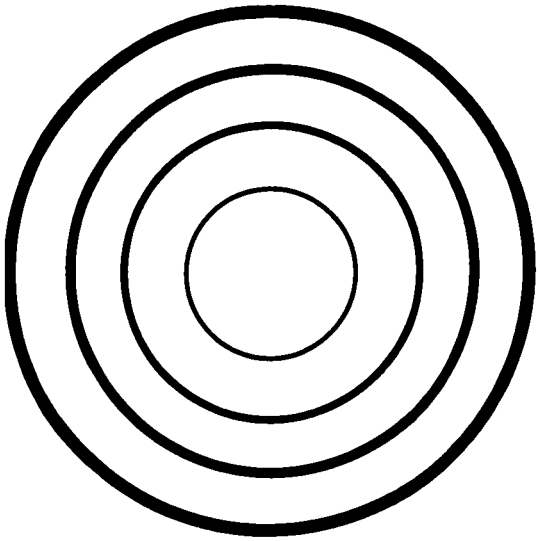

Figure 2.1: Concentric waves in a lake.

However, unlike the lake, the ear's pinna is not flat. The pinna creates different ripples depending on the direction that the sound came from. As an example, ripples of waves are shown in Figure 2.2. Each fold in the pinna creates a unique reflection, and the reflection depends on the angle at which the sound hits the ear and the frequency of the sound. For each combination of specific patterns (one from each ear) such as the one shown in Figure 2.2, our brain has learned to associate a specific direction of arrival. This is basically how sound localization works [9].

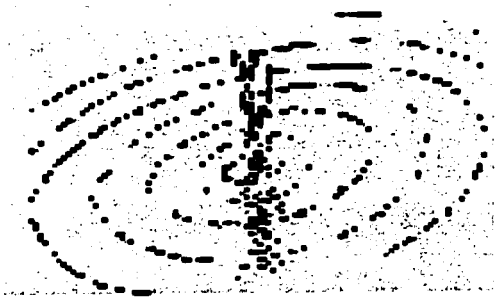

Figure 2.2: Ripples of waves on a non-flat surface.
Source from Morgan J. Dempsey " How positional 3D audio works". *http://www.vlsi.com*

2.3 Physical Variables in Acoustics

Some fundamental variables in acoustics are energy, speed of propagation and frequency. Figure 2.3 shows a case where the source of acoustic energy is a loudspeaker. Sound travels through the air at about 340 meters per second, and the cone of the loudspeaker vibrates in the air causing disturbances depending on the electrical signals reaching the loudspeaker from a sound system. Effectively, the loudspeaker converts electrical energy into sound energy, which travels through the air as waves radiating from the loudspeaker [10].

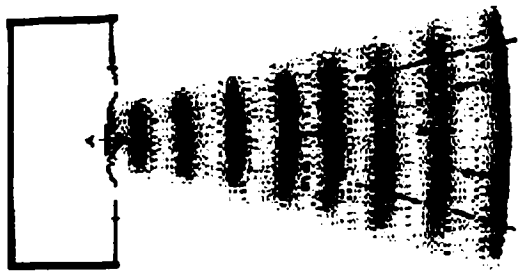

Figure 2.3: Loudspeaker emitting sound that travels through the air at about 340 meters per second.

Figure 2.4 shows the variations in air pressure over time, for a *pure sine tone* – the sound produced by a pulsating sinusoidal source. Two main physical measurements describe this wave: the amplitude and the period. The frequency of vibrations is equal to the inverse of the period. The frequency simply measures the number of waves that travel by each unit of time. It is usually measured in hertz, 1 hertz (Hz) is 1 cycle per second. The range of human hearing is approximately from 20 Hz to 20,000 Hz [10,11].

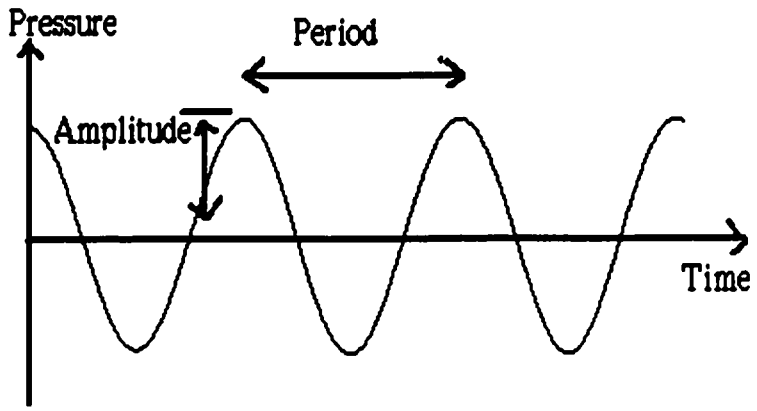

Figure 2.4: Sine wave.

2.4 Cues of Psychoacoustics

When waves reach one of our ears, they cause the eardrum to vibrate and to transmit the vibrations to the inner ear via the ossicles (Figure 2.5). In the inner ear a remarkable organ called the cochlea converts these vibrations into neural (nervous) energy, for interpretation by the higher levels of the auditory system. In the early nineteenth century, scientists started to attempt to measure the response of subjects to sounds with controlled physical characteristics. They determined that the perceptual attributes of loudness, pitch, sharpness, tonality, and roughness were strongly related to the physical variables of amplitude and frequency [9,11,12].

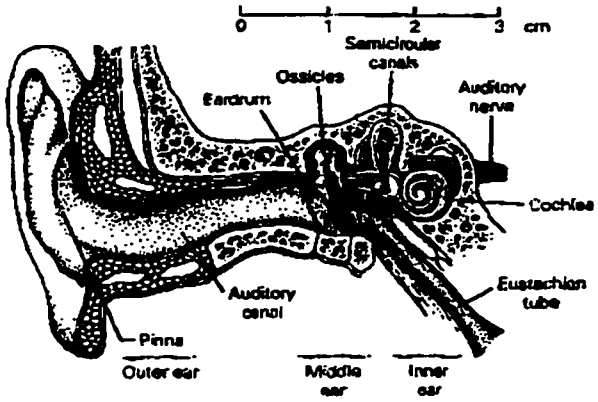

Figure 2.5: Human ear structure.
Source from http://www.online.anu.edu.au/ITA/ACAT/drw/PPofM/hearing/hearing1.html

We as humans extract a lot of information that can be retrieved from sound, and to show exactly how it is done we need to look at how sounds are perceived in the real world. To do so it is useful to break the acoustics of the real world into three major components [1,13]:

- The first component is the sound source, which is an object in the real world that emits sound waves such as human speech, car noise, bird songs, closing of doors and so on. Sound waves get created through a variety of mechanical processes. Once created, the waves usually get radiated in a certain direction. For example, a mouth radiates more sound energy in the direction where the face is pointing than to the side of the face.
- The second component is the acoustic environment. Once a sound wave has been emitted, it travels through an environment where several things can happen to it. The air absorbs it. The amount of absorption depends on factors like wind and air humidity. The sound can directly travel to a listener (direct path), bounce off of an object twice before it reaches the listener (second order reflection path) and so on.

- The third component is the listener; this is a sound-receiving object. The listener uses acoustic cues from the two ears to interpret the sound waves that arrive, and to extract information about the sound sources and the environment.

As the sound waves encounter the outer ear flap (the pinna), they interact with the complex convoluted folds and cavities of the ear, as shown in Figure 2.6. These support different resonant modes depending on the direction of arrival of the wave, and these resonances and anti-resonances can amplify or suppress the incoming acoustic signals at certain associated frequencies [14].

For example, the main central cavity in the pinna, known as the concha, makes a major contribution at around 5.1 kHz to the total effective resonance of the outer ear, boosting the incoming signals by around 10 to 12 dB at this frequency. Other resonances occur, which are optimally stimulated at specific angles of incidence. Consequently, the head and pinna (and auditory canal) can be considered as spatial encoders of the arriving sound waves, which are then spatially decoded by the two aural cortices in the brain [14]. The brain thus knows how to figure out a fairly accurate location of the sound in 3-D space by receiving a signal that has been filtered in a way that is unique to the sound source's position relative to the listener.

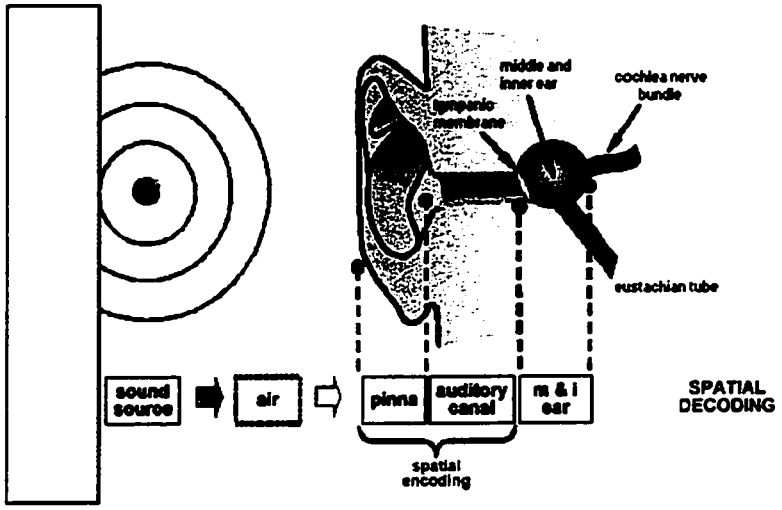

Figure 2.6: Free-field acoustic transmission pathway into the ear.
Source from http://www.headwize.com/tech/sibbald_tech.htm

2.4.1 Coordinate System

Since HRTFs for a specific direction are identified by the elevation and the azimuth of that direction, a coordinate system is needed to specify the location of a sound source relative to the listener. The head-centered rectangular-coordinate system is shown in Figure 2.7 [15]. The x-axis goes through the right ear and is parallel to the inter-aural axis, the y-axis points straight ahead, and the z-axis is vertical. This defines three standard planes, the x-y which is called horizontal plane, the x-z that is called frontal plane, and the y-z plane, which is called median plane (also called the mid-sagittal plane). Clearly, the horizontal plane x-y defines up/down separation, the frontal plane x-z defines front/back separation, and the median plane y-z defines right/left separation. A spherical coordinate system is typically used since the head is roughly spherical. The standard coordinates are azimuth, elevation and range (or distance) [1]. Unfortunately, there is

more than one way to define these coordinates, and different people define them in different ways.

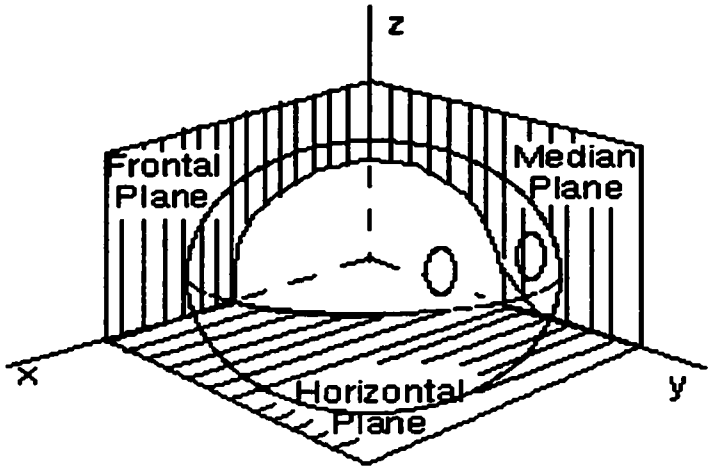

Figure 2.7: 3-D Planes (X, Y, Z).
Source from 1996-1997 Richard O. Duda http://www-engr.sjsu.edu/~knapp/HCIROD3D/3D_psych/coord.htm

The vertical-polar coordinate system (shown below on left of Figure 2.8) is the most popular. It measures the azimuth θ as the angle from the median plane to a vertical plane containing the source and the z-axis, and then measures the elevation ϕ as the angle up from the horizontal plane. With this choice, surfaces of constant azimuth are planes through the z-axis, and surfaces of constant elevation are cones concentric about the z-axis [16,17].

An important alternative is the interaural-polar coordinate system, shown below on Figure 2.8 at the right. This measures the elevation ϕ as the angle from the horizontal plane to a plane defined by the source and the x-axis, which is the interaural axis; the azimuth θ is then measured as the angle over from the median plane. With this choice, surfaces of constant elevation are planes through the interaural axis, and surfaces of

constant azimuth are cones concentric with the interaural axis. The vertical-polar system is definitely more convenient for describing sources that are confined to the horizontal plane, since one merely has to specify the azimuth as an angle between -180° and +180°. With the interaural-polar system, the azimuth is always between -90° and +90°; and the front/back distinction must be specified by the elevation, which is 0° for sources in the front horizontal plane, and 180° (or -180°) for sources in the back [17,18].

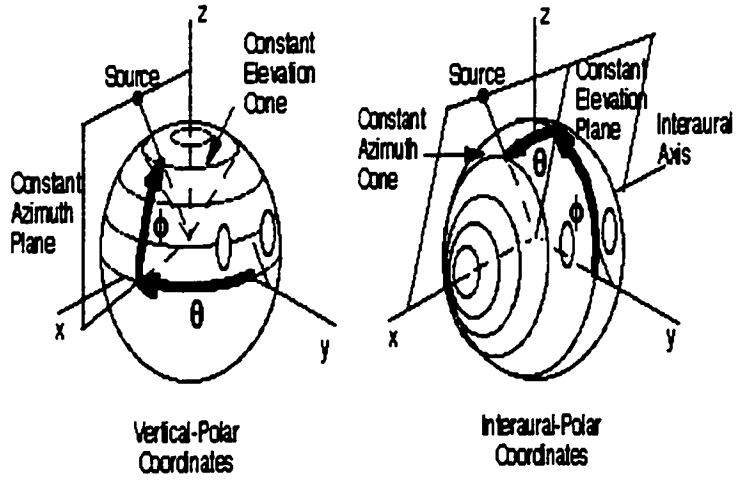

Figure 2.8: 3-D Spherical Coordinates. Source from 1996-1997 Richard O. Duda http://www-engr.sjsu.edu/~knapp/HCIROD3D/3D_psych/coord.htm

2.4.2 Horizontal Identification and Azimuth Cues

One of the pioneers in spatial hearing research was John Strutt, who is better known as Lord Rayleigh. About 100 years ago, he developed his so-called Duplex Theory. According to this theory, there are two primary cues for azimuth: Interaural Time Differences (ITDs) and Interaural Intensity Differences (IIDs) [1,19].

The first cue the brain uses to locate sounds is the time difference between the sound reaching one ear and then the other ear. Figure 2.9 shows how the ear that receives sound

first is always closer to the source [13,16,18]. If there is a long time delay before the sound reaches the more distant ear, then the brain infers that the sound comes at a great angle from this more distant ear to the sound source. It is also worth noting that if we use an interaural-polar coordinate system and hold the azimuth constant, then we obtain a constant value for the ITD. Thus, there is a simple one-to-one correspondence between the ITD and the cone of constant azimuth, which is sometimes called the "cone of confusion" [16]. This is not the case for the vertical-polar system, where a given ITD may correspond to different azimuths. Finally, note that the ITD alone only constrains the source to be somewhere on the cone of confusion. In particular, it is not sufficient for determining whether the source is in front or in back. ITD cues are thus inherently ambiguous. For pure tones, only head movements can resolve this ambiguity. But for complex sounds the effects of the pinna can resolve the ambiguity.

To distinguish a time delay between the ears, the brain must be able to discern and identify differences between the sounds as it reaches the two ears. Human heads are about seven inches wide at the ears. Sound travels in air at about c=340 m/s and humans hear sounds between f = 20 and 20,000 Hz in frequency, with wavelength λ being directly related to frequency according to:

$$f = c/\lambda \tag{2.1}$$

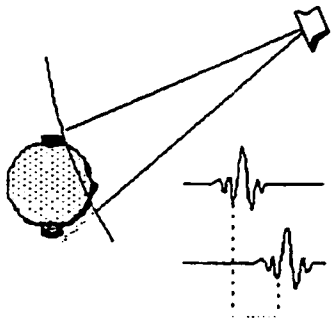

Figure 2.9: Interaural Time Differences (ITDs). Source from
http://www.headwize.com/tech/aureall_tech.htm

As shown in Figure 2.10, Lord Rayleigh had a simple explanation for the ITD [15,19]. Sound travels at a speed c of about 340 m/s. Consider a sound wave from a distant source that strikes a spherical head of radius "a" from a direction specified by the azimuth angle "θ". Clearly, the sound arrives at the right ear before the left, since it has to travel the extra distance a θ +a sinθ to reach the left ear. Dividing that by the speed of sound, we obtain the following simple and accurate formula for the interaural time difference:

$$\text{ITD} = \frac{a}{c}(\theta + \sin\theta), \quad -\pi/2 \le \theta \le \pi/2 \qquad (2.2)$$

Thus, the ITD is zero when the source is directly ahead (θ = 0), and is a maximum of $\pm\frac{a}{c}$ (π/2+1) when the source is off to one side (θ = -π/2 or π/2). This represents a difference of arrival time of about 0.7 ms for a typical size human head.

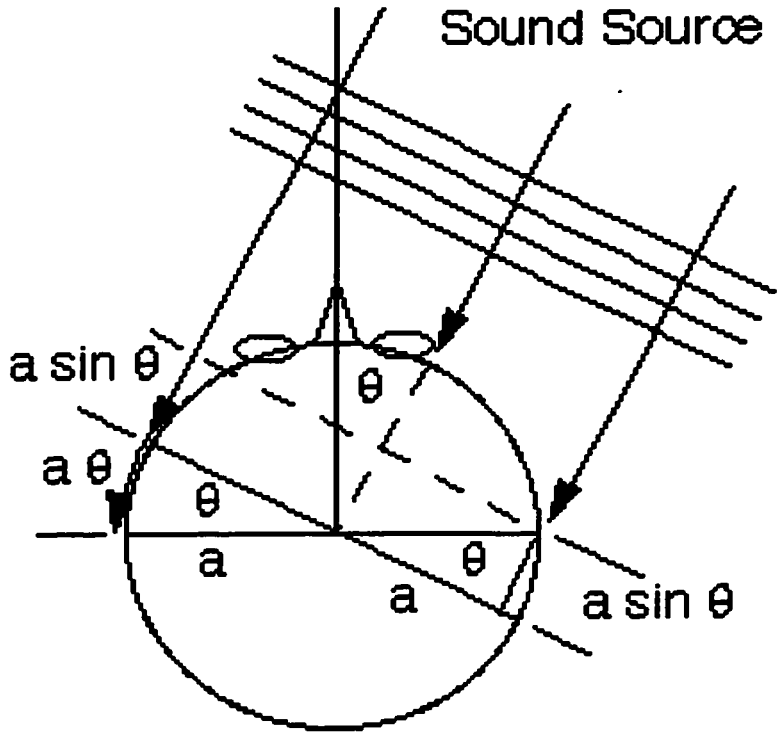

Figure 2.10: Lord Rayleigh's simplified model. Source from 96-97 Richard O. Duda http://www-engr.sjsu.edu/~knapp/HCIROD3D/3D_psych/azimuth.htm

Figure 2.11 shows that a second cue for determining horizontal direction is the sound intensity. Since the head creates an audio shadow, then the sound waves that come from one side of the head are attenuated after passing through the head to reach the far ear.

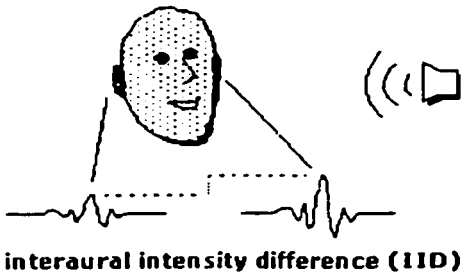

Figure 2.11: Interaural Intensity Differences (IIDs). Source from (http://www-engr.sjsu.edu/~knapp/HCIROD3D/3D_psych/azimuth.htm)

Lord Rayleigh had observed that the incident sound waves are diffracted by the head. He actually solved the wave equation to show how a plane wave is diffracted by a rigid sphere. His solution showed that in addition to the time difference there was also a significant difference between the signal levels at the two ears: the Interaural Intensity Differences (IIDs). These IIDs are highly frequency dependent, since the diffraction by the head varies with frequency. At low frequencies, where the wavelength of the sound is long relative to the head diameter, there is hardly any difference in sound pressure at the two ears. However, at high frequencies, where the wavelength is short, there may well be a 20-dB or greater difference, which is called the head-shadow effect, where the far ear is in the sound shadow of the head. Just like ITDs, IIDs cannot generally help to distinguish whether the sound is above or below the horizontal plane of the ears, and they can not generally help to distinguish between front and back [16,17].

2.4.3 Vertical Identification and Elevation Cues

The primary cues for azimuth are binaural (we need two ears to extract them), but the primary cues for elevation are often said to be monaural (we can extract them with only one ear) [20]. It stems from the fact that our outer ear or pinna acts like an acoustic antenna. Its resonant cavities amplify some frequencies, and its geometry leads to interference effects that attenuate other frequencies. Moreover, its frequency response is directionally dependent. For any given angle of elevation, some frequencies will be enhanced, while others will be greatly reduced. The brain uses the high frequency part of the sound spectrum to locate the vertical position of a sound source.

Figure 2.12 shows measured frequency responses for two different sound directions of arrival [21]. In each case we see that there are two paths from the source to the ear canal, one is the direct path and the other is a longer path following a reflection from the pinna or concha. At moderately low frequencies, the pinna essentially collects additional sound energy, and the signals from the two paths arrive in phase. However, at high frequencies, the delayed signal can be out of phase with the direct signal, and destructive interference occurs. The greatest destructive interference occurs when the difference in path length d is half a wavelength.

$$f = c / 2d \tag{2.3}$$

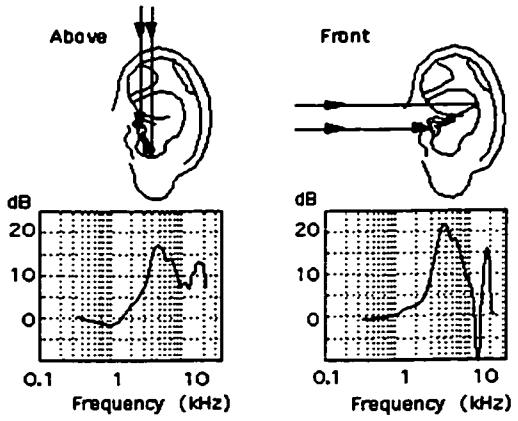

Figure 2.12:Frequency responses for two different directions of arrival. Source from " http://www-engr.sjsu.edu/~knapp/HCIROD3D/3D_psych/elev.htm"

In the right side of Figure 2.12, we see that a "pinna notch" occurs around 10 kHz. With typical values for d, the notch frequency is usually in the 6-kHz to 16-kHz range. Since the pinna is a more effective reflector for sounds coming from the front than for sounds coming from above [21,22,23], the resulting notch is much more pronounced for sources in front than for sources above.

2.4.4 Front-Back Identification

The shape of our heads, pinna, and our slightly forward facing ears also work as an audio frequency filter useful for front to back identification. Frequencies between 250 and 500 Hz and above 4000 Hz are relatively less intense when the source is behind us. On the other hand, frequencies between 800 and 1800 Hz are relatively less intense when the source is in front of us. Our memory of common sounds assists the brain in its frequency evaluations [10]. Unconsciously, we have learned the frequency spectrum content of common sounds. When we hear a sound, we compare it to the frequency spectrum of a reference sound in our memory. But sometimes the front or back location is still unclear. Without thinking, people then turn their heads to align one ear toward the sound source so that the sound intensity is highest in one ear [24].

2.4.5 Cues found and not found in Head-related Transfer Functions

We have discussed in the previous sections basic cues, which together form much of the basics of psychoacoustics localization. First there are the Interaural Time Differences (ITDs). Secondly there are the Interaural Intensity Differences (IIDs). Third there is the outer ear's response, which attenuates the spectrum of an incoming sound depending on its directions. A fourth cue can also be mentioned here: shoulder echoes. This cue can also be used for vertical identification, and it can actually be combined with the outer ear's response. Since the HRTFs are by definition the transfer functions between a sound source and the sounds received either at the entrance of the two ear canals or at the two eardrums, these four cues are thus embedded in each individual's HRTFs [25]. The

software system that will be developed in this thesis will be based on HRTF filtering, and therefore the effect of these four cues will be included in the system.

However, there are some audio cues that are not included in the HRTFs. Some cues that are not included in HRTFs are range (distance) cues and reflections (reverberation). Range cues will be discussed first. When it comes to localizing a source, in general we are best at estimating azimuth, next best at estimating elevation, and worst at estimating range or distance. In a similar fashion, the cues for azimuth are quite well understood, the cues for elevation are less well understood, and the cues for range are least well understood. Some of the range cues are [15,17,19]: loudness, ratio of direct to reverberant sound, Doppler motion effect, motion parallax, and excess interaural intensity difference. These cues will not be discussed in this thesis, because they will not be implemented in the 3-D audio systems to be developed.

Sound reflections are another very important effect in an acoustical environment. To a certain extent we are able to hear the difference in time of arrival and location between the direct path, the first order, and the n-th order reflections as shown in Figure 2.13 [13].

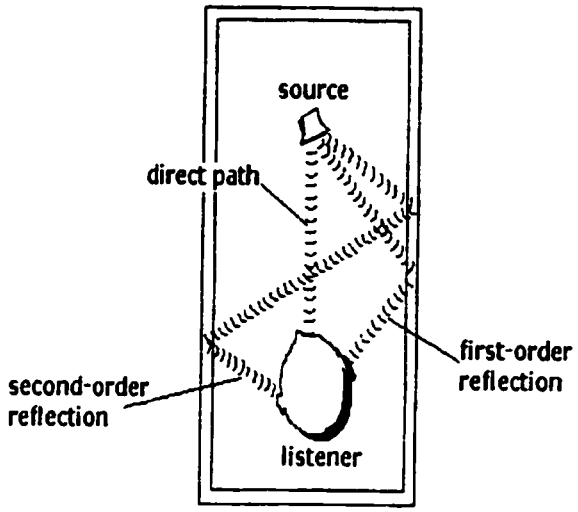

Figure 2.13: Typical sound field with a source, an environment, and a listener.
Source from (http://www.headwize.com/tech/aureal1_tech.htm)

Usually, we do not realize how much of the sound that we hear comes from reflections due to surrounding surfaces. A significant amount of energy is reflected by the ground and by surrounding structures. We only notice these reflections when the time delay gets longer than 30ms, and perceive them as echoes. In Chapter 3, it will be described how the effects of the early reflections in a virtual rectangular room were included in the offline 3-D sound generation system.

2.5 Measurement of the Head-related Transfer Functions

As previously mentioned in this thesis, the transformation of sound from a point in space to the ear canal can be measured accurately: the measurements are called head-related transfer functions (HRTFs) if they are expressed in the frequency domain, and hear-related impulse responses (HRIRs) if they are expressed in the time domain. The measurements are taken by inserting miniature microphones either at the entrance of the

ear canal or at the eardrum of a human subject or a manikin [25,26]. A measurement signal is played by a loudspeaker and the microphone signals are recorded. A computer can then be used to derive a pair of HRTFs or HRIRs (for the left and right ears) corresponding to each specific sound source location, using correlation techniques. This process is shown in Figure 2.14. The measurement procedure is repeated for many locations of the sound source relative to the head, resulting in a database of hundreds of HRTFs that describe the sound transformation characteristics of a particular head.

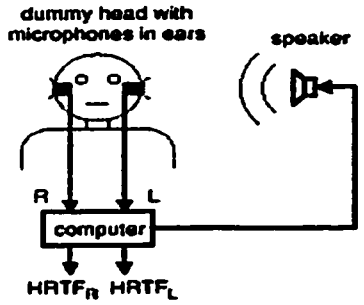

Figure 1. Measurement of HRTFs.

Figure 2.14: Measurement of HRTFs.
Source from "http://www.headwize.com/tech/gardner_tech.htm"

The HRIRs used in this thesis were measured from a Knowles Electronic Manikin for Acoustic Research (KEMAR). The measurements were made by a team at MIT, in MIT's anechoic chamber shown in Figure 2.15. The KEMAR is an anthropomorphic manikin whose dimensions were designed to equal those of a median human. The pinnas used were molded from human pinna. To describe the MIT measurements, the vertical-polar coordinates system will be used. This coordinate system will be used in the rest of the thesis. The spherical space around the KEMAR was sampled at elevation from –40 degrees to +90 degrees (directly over head). At each elevation a full 360 degrees of

azimuth was sampled in equal sized increments. Table 2.1 shows the number of samples and azimuth increments at each elevation. A total of 710 locations were sampled [24].

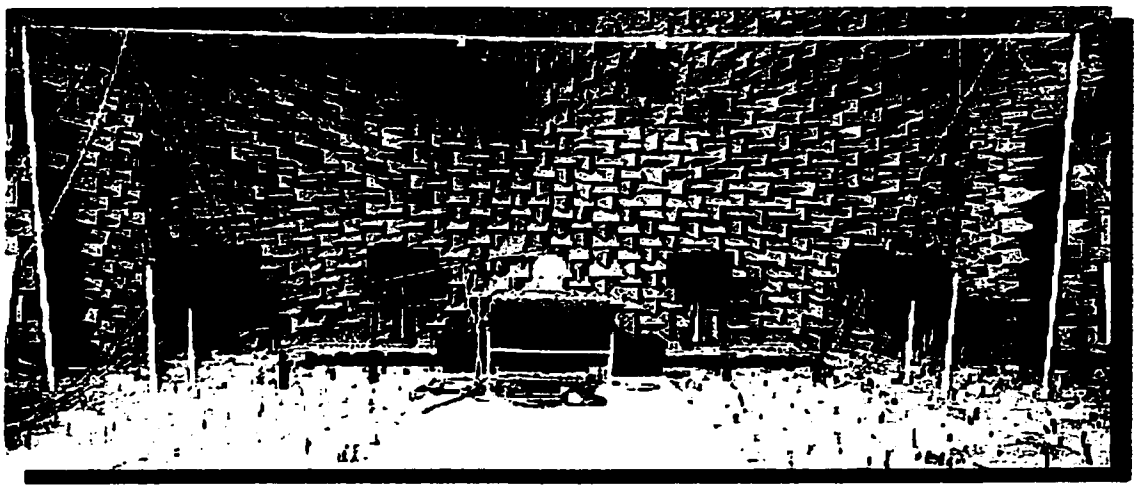

Figure 2.15:Anechoic Chamber at MIT Media Lab.
Source from "http://www.isvr.co.uk/faciliti/lg_anech.htm"

Elevation (degrees)	Number of AZIMUTHAL Measurements	Azimuth Increment (degrees)
-40	56	6.43
-30	60	6.00
-20	72	5.00
-10	72	5.00
0	72	5.00
10	72	5.00
20	72	5.00
30	60	6.00
40	56	6.43
50	45	8.00
60	36	10.00
70	24	15.00
80	12	30.00
90	1	0.0

Table 2.1: Number of azimuthal measurements and azimuth increment at each elevation in the MIT HRIR database (vertical-polar coordinates). Source from ftp://sound.media.mit.edu/pub/Data/KEMAR/hrtfdoc.txt

Measurements at MIT were made using a Macintosh Quadra computer equipped with an Audio Media II DSP card. Figure 2.16 shows all the components that were originally included in the original HRIR impulse responses measured by the MIT researchers. The recording system is consisting of 16-bit stereo D/A and A/D converters that operate at a 44.1 kHz sampling rate, with anti-aliasing filters. One of the audio output channels was sent to an amplifier, which drove a Realistic Optimus Pro 7 speaker. The KEMAR, Knowles Electronics model DB-4004, was equipped with model DB-061 left pinna, model DB-065 right pinna (we only used the measurements from the "small" left pinna in this thesis), Etymotic ER-11 microphones and Etymotic ER-11 preamplifiers. The outputs of the microphone preamplifiers were connected to the stereo inputs of the Audio Media card [24].

Ideally, HRIRs should only contain the propagation delay in the anechoic room and the PINNA component of Figure 2.16 (which includes the pinna, the shoulders and the head shadow effects). In particular, the effect of the ear canal should in general not be included in the HRIRs, since there will be a physical ear canal in any playback system using HRIRs. Including the ear canal in the HRIRs could thus lead to situations where the effect of the ear canal appears twice in the sounds that reach the eardrums, and this may cause distortion and a decrease of the localization performance of the HRIRs.

As it can be seen from Figure 2.16, the original measured HRIRs contain more elements than the ideal HRIRs. In an attempt to remove the effect of the undesired elements from the measured HRIRs, the MIT team has also produced a set of HRIRs called “diffuse”.

They did so by averaging the HRIRs over all spatial positions, hoping that this would only leave the undesirable spatially-independent components in Figure 2.16, and then they removed these components by filtering the original measurements with the inverse response of those spatially-independent components, producing the "diffuse" set of HRIRs. In this thesis, we will experiment with both the original HRIR measurements (called "compact") and the "diffuse" set of HRIRs, just to see if one performs better than the other with our setup.

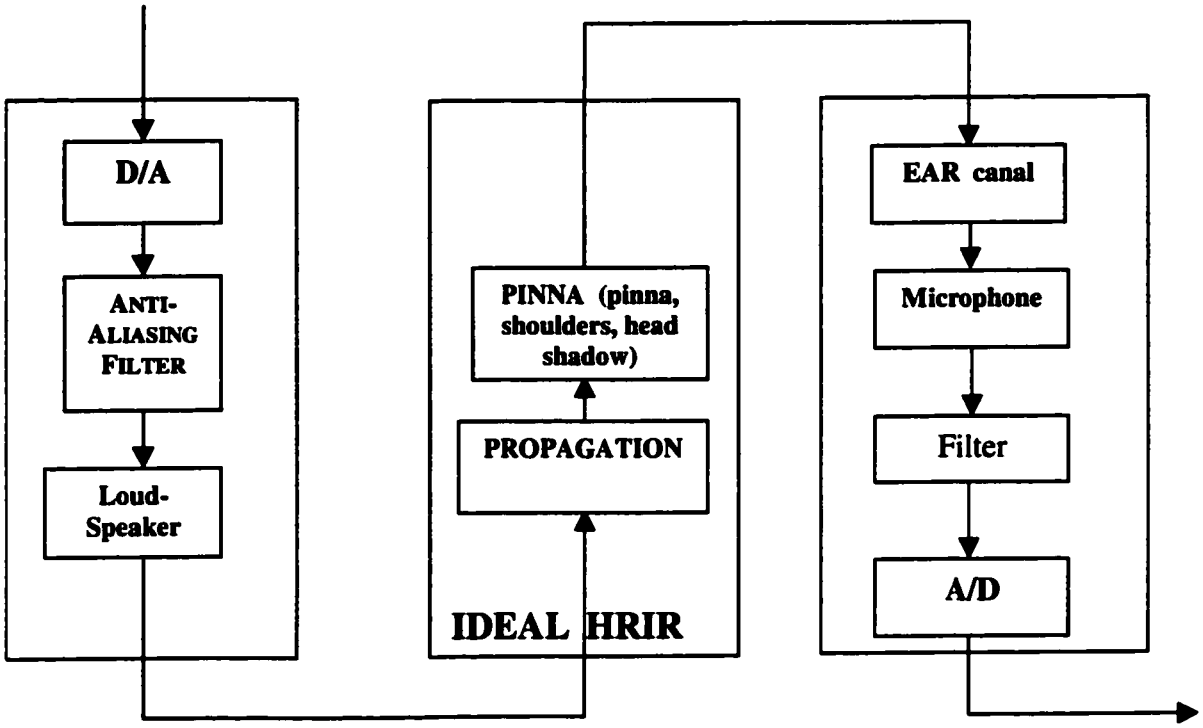

Figure 2.16: HRIR Measurements.

2.6 Synthesis of 3-D Sounds using HRTFs

The process of filtering a mono sound source with left and right HRTF filters is called binaural synthesis, and this is further illustrated in Figure 2.17:

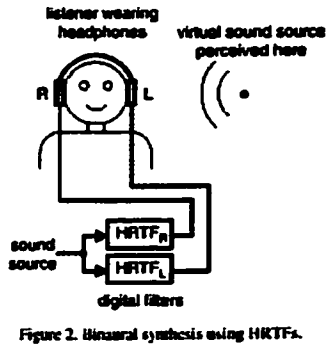

Figure 2.17: Binaural synthesis using HRTFs.
Source from "http://www.headwize.com/tech/gardner_tech.htm"

Binaural synthesis works well when the listener's own HRTFs are used to synthesize the localization cues [16,20,23,24]. However, measuring HRTFs is a complicated procedure, so 3-D audio systems typically use only a few sets of HRTFs previously measured from a few particular human or manikin subjects.

Localization performance generally suffers when a listener listens to directional cues synthesized from HRTFs measured from a different head [25], called non-individualized HRTFs. This is because human heads are all of different sizes and shapes, and there is also a great variation in the sizes and shapes of individual pinna. This means that every individual has a different set of directional cues. The greatest differences are in the amplitude and phase transformations at high frequencies caused by the pinna. It is thus clear that we become accustomed to localizing with our own ears, and thus our localization abilities will be diminished when listening through another person's ears. Our uniqueness as individuals is the source of the greatest limitation of binaural 3-D sound technology.

The use of non-individualized HRTFs produces mainly two particular kinds of localization errors in 3-D audio systems: front/back confusions and elevation errors [24]. Front/back confusion occurs when the listener perceives the sound to be in the front when it should be in the back, and vice-versa. When 3-D audio is reproduced over frontal loudspeakers, back to front confusions tend to be common, which simply means that some listeners may not be able to perceive sounds as being in the rear. In practice, this means that when panning a sound from the front, around to the side, and to the rear, the result will be perceived as a sound panning to the side and then back to the front.

Elevation errors are also common with 3-D audio systems. In practice, when a sound is moved from directly to the right to directly overhead, this may be perceived as though the sound is moving from the right to directly in front. This is a typical manifestation of elevation errors, commonly observed when using loudspeakers. Elevation performance is much better when using headphones than when using loudspeakers because the high frequency cues are more faithfully reproduced, but still the use of non-individualized HRTFs will cause some elevation errors [23,24,25].

2.7 Crosstalk and HRTF adaptation

When reproducing localization cues to a listener using HRTFs, it is important that the left and right audio channels remain separated, that is, the left ear signal should go to the listener's left ear only, and the right ear signal should go to the listener's right ear only. This is easy to achieve when the listener is using headphones. When using loudspeakers, however, there is significant "crosstalk" between each speaker and the opposite ear of the

listener. A large portion of the left speaker signal will go to the right ear of the listener, and similarly a large portion of the right speaker signal will go to the left ear of the listener. In Figure 2.18, the crosstalk paths are labeled ALR and ARL. The crosstalk severely degrades localization performance and must be eliminated when loudspeakers are used [26,27].

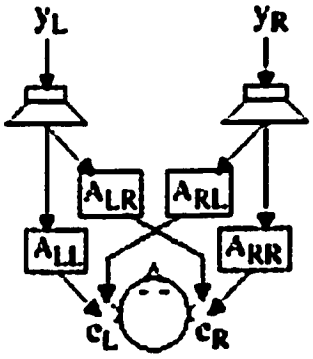

Figure 2.18: Direct and crosstalk transmission paths from loudspeakers to the ears of a listener. Source from "http://www.headwize.com/tech/gardner_tech.htm"

It is possible to build an elaborate digital filter, called a "crosstalk canceller," that eliminates crosstalk. However, this introduces extra computational complexity and may require adaptive identification of the different acoustical paths if the listener is moving. In the case where headphones are used, not only is there very little crosstalk, but also the acoustic transfer functions between loudspeakers and eardrums typically do not change as listener moves, which is a clear advantage [26,27,28].

However what does it change in both cases (loudspeakers or headphones) is the HRTF transfer functions if a virtual sound source should not move with the listener as a listener moves. This requires some head-tracking device to find the position and direction of the

listener, then the elevation and azimuth of the virtual sound source location must be computed using the listener as the origin, and finally the appropriate HRTFs must be selected and used to filter the mono input sound source signal.

In the application developed for this thesis, it is supposed that the listener is not moving, since it is expected that in a desktop virtual environment the user will be in front of a monitor and looking at the monitor. Sound sources however may be moving around the listener. This simplification obviously eliminates the need for head-tracking devices. Also, because it is not realistic to perform adaptive identification and compensation of acoustic paths in real-time on a standard PC with limited computational power, this thesis will only consider the problem of rendering 3-D sounds through headphones, where the amount of crosstalk is minimal.

2.8 Problematic of Headphone Playback with HRTF generated sounds

Some problems of headphone playback with HRTF generated 3-D sounds will be addressed in this section. Ideally, the rendering of 3-D sounds generated via HRTFs or HRIRs would be as in Figure 2.19, and individualized HRTFs would be used. As it has been mentioned before in this thesis, using individualized HRTFs is not a realistic goal at this point, because HRTFs require an extensive set of measurements. The ideal system of Figure 2.19 also supposes that the HRTFs do not already include the ear canal effect, and of course that the HRTFs do not include any component from the converters, loudspeakers, microphones, amplifiers and filters that were used during the measurements of the HRTFs.

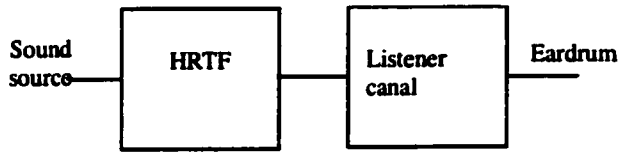

Figure 2.19: Ideal playback of HRTF generated 3-D sound.

In practice, it is not only that non-individualized transfer functions are used, but also that a system similar to the one shown in Figure 2.20 is often used. It can be seen from this figure that before a sound wave reaches the eardrum, it will be distorted by the headphone loudspeaker, and by the combined headphone cavity and listener pinna. This distortion will unavoidably cause degradation in the spatialization performance of the 3-D audio system.

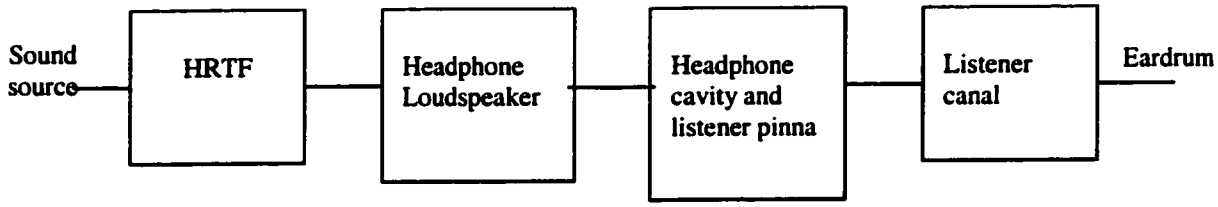

Figure 2.20: Playback of HRTF generated 3-D sound via headphones.

It is possible to perform a compensation of the headphone loudspeaker transfer function and the headphone cavity and listener pinna transfer function. This requires measuring the transfer function between a sound sent through the headphones loudspeaker and a microphone probe in the ear canal [3]. This has been investigated at one point in the thesis. However, we did not have the proper equipment to perform measurements of the transfer function (our microphone was built-in the loudspeaker and was not at the

entrance of the ear canal or at the eardrum), and the results were not conclusive (i.e. no improvement during the preliminary listening tests using the compensation). Since this was certainly not one of the main objectives of this thesis, this aspect was not investigated further.

2.9 Chapter summary

In this chapter some fundamental aspects of sound localization theory were introduced. The knowledge of those aspects is critical in order to design properly a 3-D sound system. Psychoacoustics cues were discussed, and it was mentioned which cues were found in HRTFs and which cues were not found in HRTFs. It was also mentioned and explained why this thesis deals only with binaural rendering using headphones, for listeners who do not move in a virtual space. Some problems related to generating 3-D sounds using HRTFs were also discussed.

Chapter 3 Implementation of an Offline Binaural Audio System

3.1 3-D Audio System Generating Sound Files

Now that the psychoacoustic cues found in the HRTFs have been reviewed in the previous chapter, the design of the 3-D sound system can now be explained. It has been stated that a goal of the thesis is to develop a system using headphones for listeners who do not move in a virtual space. It was decided to first develop an offline (i.e. file based, not real-time) version of the 3-D sound system. This version reads an input mono sound file, and writes a stereo 3-D sound file to be heard via headphones. The motivation for this offline version is that the programming of the sound spatialization algorithm and the real-time programming are both challenging tasks, and it is better to de-couple these two tasks to allow better or easier debugging and performance evaluation. Therefore, the real-time programming aspects are delayed until Chapter 4. Since the offline version developed in this chapter does not have any real-time constraints, it will be possible to introduce environmental effects such as propagation loss and reflections to the HRTF filtering of the 3-D sound software.

3.2 Equations and algorithms used for the 3-D sound generation

To reproduce a virtual sound source in an anechoic environment, at a specific location relative to the listener, it is possible to summarize the HRIR time-domain convolution (i.e. filtering) operation with two equations:

$$y_l[n] = \sum_{k=0}^{M-1} hrir_{l,\theta,\phi}[k]x[n-k] \tag{3.1}$$

$$y_r[n] = \sum_{k=0}^{M-1} hrir_{r,\theta,\phi}[k]x[n-k] \tag{3.2}$$

Where

$y_l[n]$ is the left channel output signal at time n,

$y_r[n]$ is the right channel output signal at time n,

$x[n]$ is the input mono source signal at time n,

$hrir_{l,\theta,\phi}[k]$ is the k^{th} coefficient of the left channel HRIR impulse response for azimuth θ and elevation ϕ, relative to the listener position and direction,

$hrir_{r,\theta,\phi}[k]$ is the k^{th} coefficient of the right channel HRIR impulse response for azimuth θ and elevation ϕ, relative to the listener position and direction, and

M is the length of the HRIR impulse responses. A value of $M = 128$ was used for the "compact" and "diffuse" MIT HRIRs sets.

The computational load of equations (3.1) and (3.2) is quite high. For example, if a sampling rate of 32 kHz is used and $M = 128$, this requires 8 192 000 multiplies per second for a real-time implementation, and almost as many adds operations. However, in

the offline version developed in this chapter, there are no real-time constraints, and computing (3.1) and (3.2) is therefore not a problem. Including additional audio cues is also possible. It was chosen to add the effect of propagation loss, propagation delay and the effect of the early reflections in a virtual rectangular room (the choice of a virtual rectangular room is justified later in this section).

To implement the effect of propagation loss, an inverse squared distance law was assumed, and the resulting equation is:

$$Sound\ Intensity = Sound\ Intensity_{ref}\left(\frac{Dist_{ref}}{Dist}\right)^2$$

(3.3).

where

Sound Intensity is the sound intensity at a specified distance *Dist*

$Sound\ Intensity_{ref}$ is the reference sound intensity at a specified reference distance $Dist_{ref}$.

To compute the propagation delay (in samples), the following equation is used:

$$Delay = \left\lfloor \frac{Dist \times F_S}{V_S} \right\rfloor \qquad (3.4)$$

where

F_S is the sampling rate of the input and output sound files,

V_S is the velocity of sound (340 meters/second was used), and

$\lfloor \rfloor$ is the "floor" or "integer" function.

To compute the early reflections, we used the method of the "mirror sources". The idea behind this method is that the response of a sound source in a room with wall reflections can be described equivalently by the response of a sound source in an anechoic room, with additional sources in anechoic-mirrored rooms used to produce the same waves as the reflected waves in the original room. Each source in the mirrored rooms must of course have the appropriate gain and delay. This technique is fairly simple when rectangular rooms are considered, and therefore in our application a virtual rectangular room was chosen. The position of the sound sources in the mirrored rooms were computed with the following equation [29]:

$$x_m = n_x x_r + (n_x \bmod 2) x_r + (-1)^{n_x} \times x_s \tag{3.5}$$

$$y_m = n_y y_r + (n_y \bmod 2) y_r + (-1)^{n_y} \times y_s \tag{3.6}$$

$$z_m = n_z z_r + (n_z \bmod 2) z_r + (-1)^{n_z} \times z_s \tag{3.7}$$

Where

x_m is the x position of the sound source in the mirrored room,

x_s is the x position of the original sound source,

x_r is the size of the original room in the x dimension,

n_x is the order of room reflections in the x dimension used to compute a specific mirrored sound source,

y_m is the y position of the sound source in the mirrored room,

y_s is the y position of the original sound source,

y_r is the size of the original room in the y dimension,

n_y is the number of room reflections in the y dimension used to compute a specific mirrored sound source,

z_m is the z position of the sound source in the mirrored room,

z_s is the z position of the original sound source,

z_r is the size of the original room in the z dimension, and

n_z is the number of room reflections in the z dimension used to compute a specific mirrored sound source, and

mod is the modulus mathematical operator.

The maximum values of n_x, n_y and n_z (i.e. the maximum order of room reflections in each of the different dimensions) was specified by the user as a common value $N_{x,y,z}$ $(=n_x=n_y=n_z)$. The distance between a sound source in a mirrored room and the listener was simply computed by:

$$Dist=\sqrt{(x_m-x_l)^2+(y_m-y_l)^2+(z_m-z_l)^2} \tag{3.8}$$

where:

x_l is the x position of the listener,

y_l is the y position of the listener, and

z_l is the z position of the listener.

The propagation delay between each sound source in a mirrored room and the listener was computed using equation (3.4). A constant frequency independent absorption coefficient α was assumed for the reflections on the virtual walls. The value of α was set by the user ($0 \leq \alpha \leq 1.0$). With this absorption coefficient, equation (3.3) for sound propagation loss becomes:

$$Sound\ Intensity = Sound\ Intensity_{ref}\left(\frac{Dist_{ref}}{Dist}\right)^2 \alpha^n \qquad (3.9).$$

where n is the total number of reflections for a specific sound source in a mirrored room (i.e. the total number of walls that need to be crossed to reach a sound source in a mirrored room).

With the knowledge of the sound intensity and propagation delay for each sound source in a mirrored room (and for the original source), and with the knowledge of the HRIRs associated with the direction of each sound source relative to the listener, it is then possible to compute "equivalent" or "non-anechoic" HRIRs that will include the effects of the early reflections [24,25]. This is done with the following equations:

$$hrir\_eq_l[m] = \sum_{n=0}^{N-1} S_n hrir_{l,n}[m - Delay_n] \qquad (3.10)$$

$$hrir\_eq_r[m] = \sum_{n=0}^{N-1} S_n hrir_{r,n}[m - Delay_n] \qquad (3.11)$$

where

S_n is the sound intensity associated with the n^{th} mirrored sound source,

$Delay_n$ is the propagation delay associated with the n^{th} mirrored sound source,

$hrir_{l,n}[m]$ is the m^{th} coefficient of the left HRIR associated with the n^{th} mirrored sound source,

$hrir_{r,n}[m]$ is the m^{th} coefficient of the right HRIR associated with the n^{th} mirrored sound source,

$hrir\_eq_l[m]$ is the m^{th} coefficient of the "equivalent" left HRIR for the original sound source and the early reflections in the virtual rectangular room,

$hrir\_eq_r[m]$ is the m^{th} coefficient of the "equivalent" right HRIR for the original sound source and the early reflections in the virtual rectangular room, and

N is the total number of mirrored sources. This is the product of the number of room reflections in the x, y, and z dimensions.

The maximum length M' of the resulting "equivalent" HRIR is much higher than the length M of the original anechoic HRIR:

$$M' = \left(\frac{N_{x,y,z}}{2} + 1 \right) \times \sqrt{x_r^2 + y_r^2 + z_r^2} \times \frac{F_S}{V_S} + M \qquad (3.12)$$

where

$N_{x,y,z}$ is the maximum number of room reflections in the x, y, and z dimensions (same number is assumed for each dimension).

Equation (3.12) first computes the maximum delay (in samples) that can occur between a sound source at the extreme corner of the furthest mirrored room and the listener, and it adds the original HRIR length M to this delay. To see how much larger M' is compared to M, a room of 6 by 6 by 4 meters and a sampling rate of 32 kHz with only 10 room

reflections in each dimension will have a resulting maximum length of M'=5425 coefficients, instead of the original $M = 128$ coefficients. M' is called the maximum length because a designer may choose to truncate it to less coefficients than its total length. The high value of M' compared to M is the reason why early reflections were not implemented in the real-time 3-D sound system to be introduced in Chapter 4. Removing the M and F_S terms in (3.12) will produce an estimate of the length of the early reverberation produced by the mirrored room method. Keeping the same room dimensions and the same number of room reflections as the previous example, we get 165 ms of early reverberation. In our application no effort was made to add the effect of the late reverberation field, which usually requires some form of approximation due to the computational complexity of the problem [1].

3.3 Main structure of the offline 3-D sound system

The main structure of the offline 3-D sound software that was developed is shown in Figure 3.1. Basically, an input mono sound file is read and the convolution of this signal with the "equivalent" HRIRs is computed. The resulting 3-D sound signals are then written in a stereo output sound file. Whenever the position of the sound source is moving, new equivalent HRIRs are computed by the application.

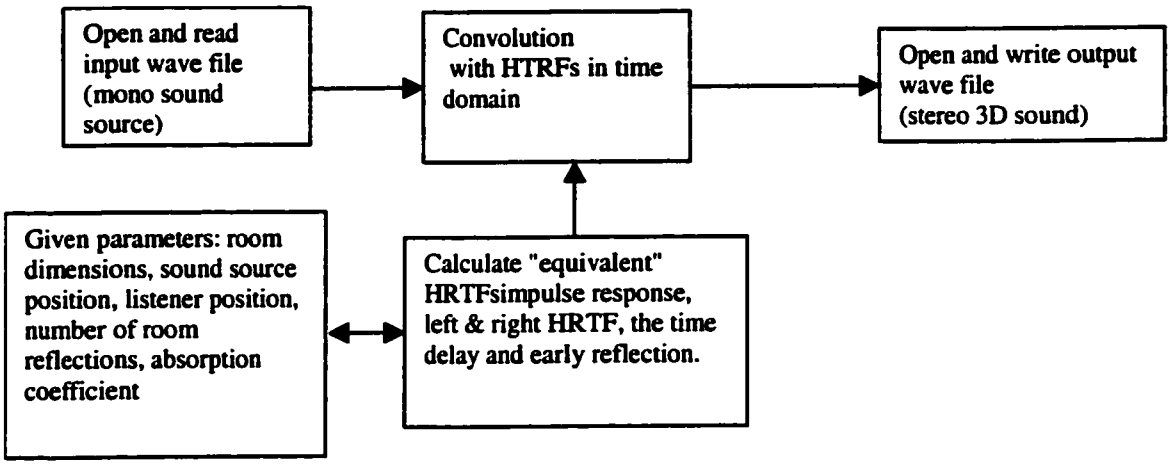

Figure 3.1: 3-D Audio system.

Since the 3-D application developed in this thesis should eventually be integrated in a virtual environment application, it was chosen to develop the 3-D sound processing module as an independent DLL module that could be called by any Windows™ application. To test the DLL module, a simple executable (.EXE) program was developed, where the following parameters were transmitted to the 3-D sound DLL: the path of the HRIRs to be used, the name of the sound source file to be filtered, the listener's position (x,y,z), the listener's head orientation (x,y only), the sound source position (x,y,z), the time duration of the sound source, the wall reflectivity coefficient, and the maximum number of room reflections in each dimension (same number for all dimensions). Note that it is assumed that the listener is looking in the (x,y) plane, but this does not mean at all that sound sources have to be located in that plane. For the room dimensions, the listener position and the sound source positions, the distances are in meters, while seconds are used as the unit for the time duration of the sound sources. Source code for the offline 3-D sound software version of Chapter 3 is not included in this thesis, since the most significant application developed in the thesis is the real-time

3-D sound generator of Chapter 4, and the source code for this application is included in the Appendix.

3.4 WAVE Files

A significant challenge in the offline 3-D sound application that was developed was the programming of the algorithms for the generation of the 3-D sound, and this has been described in section 3.2. Another challenge for the development of the offline 3-D sound system was to properly read and write WAVE sound files, and this section will provide details on this format and how we implemented the reading and writing of the sound files.

The "WAVE" File Format is a standardized file format for storing digital audio (waveform) data. It supports a variety of bit resolutions, sample rates, and audio channels. This format is very popular in PC platforms, and is widely used in professional programs that process digital audio waveforms. The overall structure of the file is based on the interchange file format (IFF). Microsoft has defined a general file format called RIFF (Resource Interchange File Format). RIFF files are organized as a collection of nested "chunks". Tags within the RIFF files identify the contents. The most common type of RIFF files are "WAVE" files [31,32], and this is the format that was used for the off-line 3-D audio generator described in this chapter.

3.4.1 WAVE File Structure

A WAVE file is a particular kind of RIFF file, and every RIFF file must begins with the characters RIFF. In a WAVE file, there is a required Format ("fmt ") chunk that contains important parameters describing the waveform, such as its sampling rate. The Data chunk, which contains the actual waveform data, is also required. All other chunks are optional. Figure 3.2 shows a graphical overview of an example, minimal WAVE file. It consists of the RIFF header, the file size, the WAVE header, and the two required chunks: a Format and a Data Chunk [33].

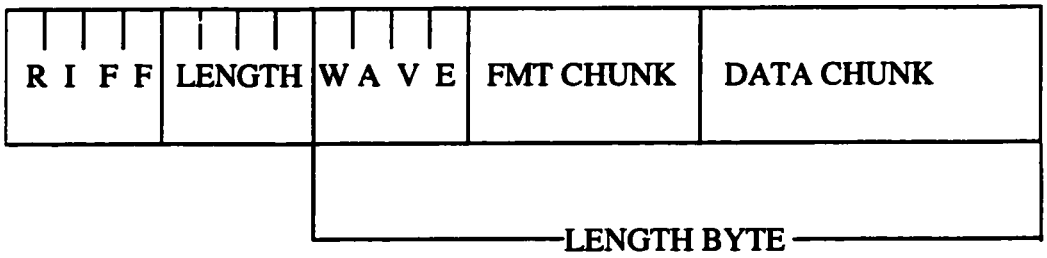

Figure 3.2: WAVE File Structure.

Most WAVE files have the same basic structure, and most programs treat WAVE files as having a fixed header with the format shown in Table 3.1:

Size in bytes	Description
4	"RIFF" header
4	Total file size in bytes minus 8
4	WAVE header
4	Chunk "fmt " header
4	"fmt " chunk length: usually 16 bytes
16	Sampling rate, nb. channels, coding type, nb. bits/sample
4	Chunk "data" header
4	Length (in bytes) of data chunk
variable	Actual sound samples

Table 3.1. WAVE File Format.

3.4.2 Format Chunk

The Format (fmt) chunk describes fundamental parameters of the waveform data such as the sample rate, the bit resolution, and how many channels of digital audio are stored in the WAVE file. In our application, a data type was defined for the format chunk, as shown below:

```
#define FormatID 'fmt '  /* chunkID for Format Chunk. NOTE: There is a space at the
end of this ID. */
typedef struct {
  ID          ChunkId;
  long        ChunkSize;
  short       wFormatTag;
  unsigned short wChannels;
  unsigned long  dwSamplesPerSec;
  unsigned long  dwAvgBytesPerSec;
  unsigned short wBlockAlign;
  unsigned short wBitsPerSample;
/* Note: there may be additional fields here, depending upon wFormatTag. */
} FormatChunk;
```

The chunkSize field is the number of bytes in the chunk. This does not include the 8 bytes used by the ID and Size fields. For the Format Chunk, ChunkSize may vary

according to what "format" of WAVE file is specified. The wFormatTag indicates whether compression is used when storing the data. If compression is used wFormatTag is equal to 1. Furthermore, compressed formats must have a Fact chunk that contains an unsigned long indicating the size (in sample points) of the waveform after it has been decompressed [33].

In this thesis there is no compression used, we set wFormatTag = 0, then there are no further fields. The wChannels field contains the number of audio channels for the sound. A value of 1 means monophonic sound, we set 2 for stereo sound. The dwSamplesPerSec field is the sample rate at which the sound is to be played back in sample frames per second (Hertz). The standard rates are 11025, 22050, 32000 and 44100 Hz. The dwAvgBytesPerSec field indicates how many bytes are played every second.

The wBlockAlign is the size of a sample frame, in terms of bytes. A sample frame for a 16-bit mono wave is 2 bytes. A sample frame for a 16-bit stereo wave is 4 bytes, etc. The wBitsPerSample field indicates the bit resolution of a sample point, a 16-bit waveform would have wBitsPerSample = 16. Only one Format Chunk is required in every WAVE file.

3.4.3 Data Chunk

The Data (data) chunk contains the actual sample frames, all channels of waveform data [33]. Below is the C++ code that was used for the data declaration structure of the data chunk:

```
#define DataID 'data' /* chunk ID for data Chunk */
typedef struct {
 ID          ChunkID;
 long        ChunkSize;
 unsigned char  waveformData[ ];
} DataChunk;
```

ChunkSize is the number of bytes in the chunk, not counting the 8 bytes used by the ID and Size fields, nor any possible padding bytes needed to make the chunk an even size. This means that chunkSize is the number of remaining bytes in the chunk after the chunkSize field, not counting any trailing padding bytes. The waveform Data array contains the actual waveform data. The data is arranged into sample frames, where the left channel sample is first and the right channel sample is second in the stereo files. The number of sample frames in waveform Data is determined by dividing this chunkSize by the Format chunk's wBlockAlign.

3.4.4 WAVE file reading and writing

The following flowcharts describe the subroutine that was used to read the input WAVE file header and advance the file pointer to the data values in the file. The data format of the input WAVE file is also checked to ensure it is the type that this program can process as input file: a 16 bits/sample, mono, PCM coded sound file.

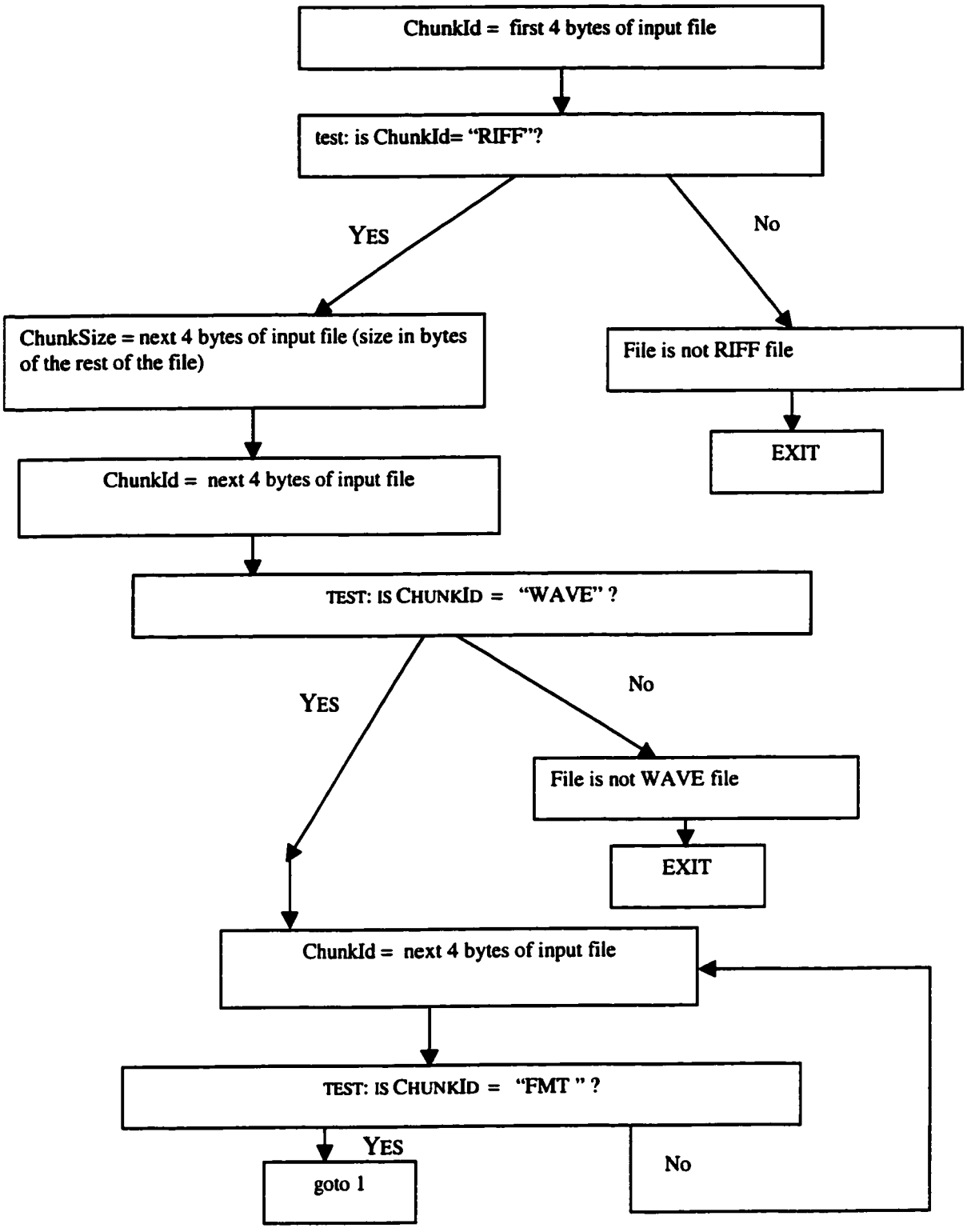

Figure 3.3: Flowchart of WAVE file reading, part A.

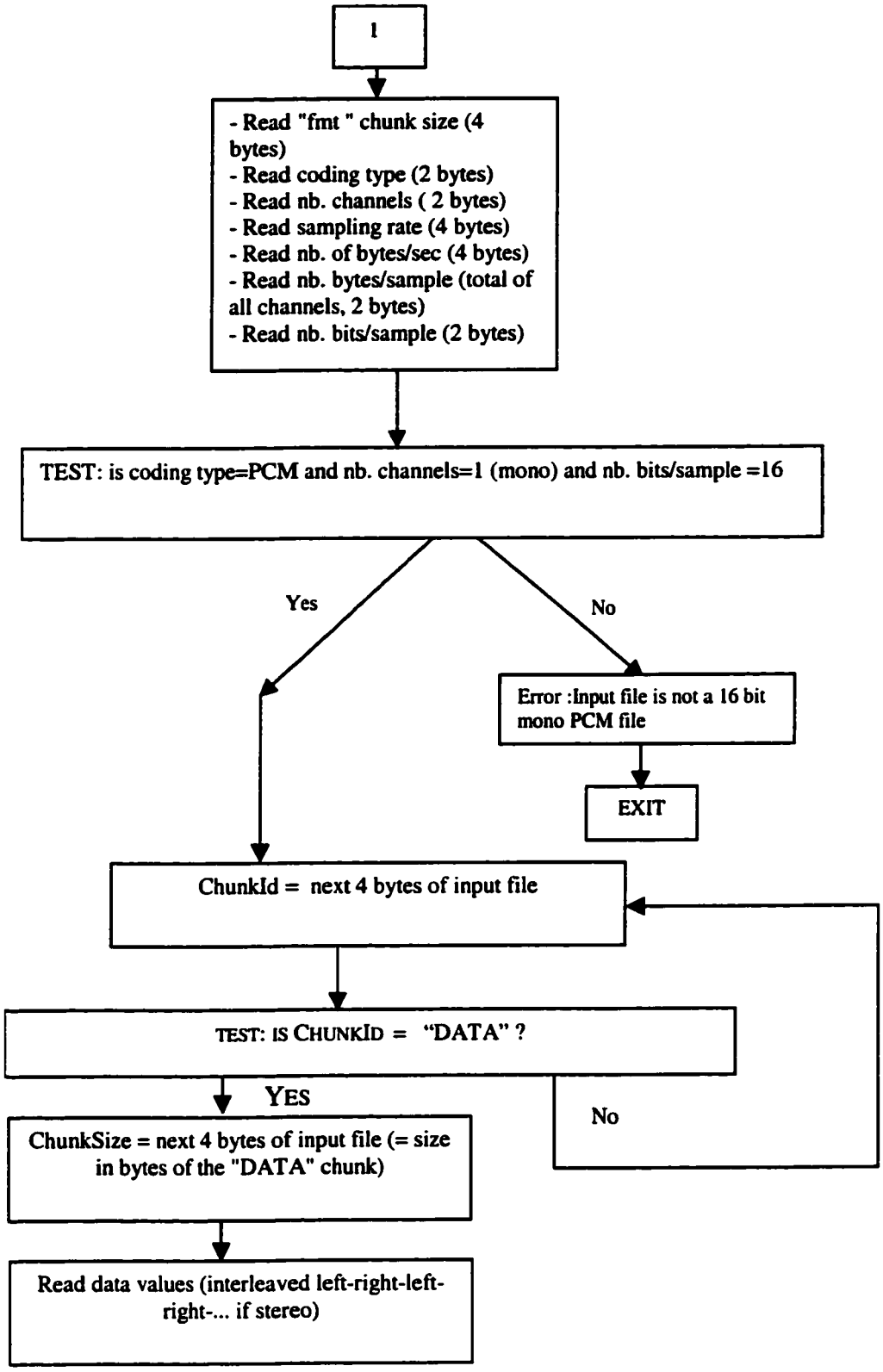

Figure 3.4: Flowchart of WAVE file reading, part B.

The information that was read from the input sound file (coding type, number of bits/sample, sample rate, etc.) is also used to create the header for the output sound file. The only fields that are changed are the fields related to the number of channels, which goes from one (mono) in the input source file to two (stereo) in the output 3-D sound file.

3.5 Listening tests using Compact and Diffuse MIT HRIR sets

Informal listening tests were performed to validate the software that was developed and to roughly evaluate the spatialization performance of the HRTF sets that were used. As mentioned previously in the thesis, two distinct sets of HRTFs were used: the "compact" set and the "diffuse" set from MIT's KEMAR measurements. It should be noted that the software that was developed could be used with just any set of HRTF, but those two sets were the only ones we had available. Since our program was not HRTF dependent, the spatialization performance was not of critical importance: it would be very easy to replace the HRTFs that we used if another set of HRTFs should provide a better performance. The listening tests were done by the candidate and his supervisor. They were performed using various low-cost open-type headphones, and occasionally with closed-type headphones. It was found that the type of headphones did not have a significant impact in our experiments, so it was decided to use low-cost open-type headphones, since they are commonly found with multimedia desktop computers. The tests did not involve a detailed listening procedure, because this was clearly not a main objective of this thesis. Such detailed procedures would involve the comparison of the sound position perceived by several listeners with the azimuth and elevation of the HRIRs, for several directions and for anechoic/non-anechoic listening environments.

The testing procedure was simply to play a 3-D sound that goes through a circular path in a virtual room, and to verify that when headphones are used with the developed 3-D sound system, the virtual sound source seem to follow the specified path. We found no

real difference in performance between the two sets of HRTFs, probably because in both cases we were far from the ideal system of individualized HRTF and individualized compensation (in fact we did not use any compensation at all, as mentioned in Chapter 2). Nevertheless, the performance of the system for a sound source that turns around a listener's head was found to be quite impressive. One effect that should be improved is the "clicks" that we heard when a sound source changes position. This is because in this case we switched abruptly from one pair of HRTFs to another pair of HRTFs without any smooth transition. A solution to improve this would be to progressively decrease the gain of the "old" HRTF pair and increase the gain of the "new" HRTF pair (this is usually called cross-fading), but this was not implemented.

3.6 Chapter summary

In this chapter, the development of an offline 3-D sound rendering system was explained. The equations and algorithms required for the 3-D sound generation (including directivity of sound and early reflections in a virtual rectangular room) were first described. Then the global software structure of the offline system was described, and finally some details on the reading and writing of WAVE files in the application were provided. Some listening tests were performed to validate the spatialization capabilities of the application. As mentioned in the methodology part of Chapter 1, the main goal of the offline system was to validate the 3-D sound algorithms that were to be used. Another goal was to add some effects (such as early reflections) that cannot be added in the real-time version of the application, because of the real-time constraint. Now that the algorithms for the 3-D sound have been validated, a real-time version of the application can be developed, and this is addressed in the next chapter.

Chapter 4 Real-time Implementation

4.1 Playing sound in real-time in a Windows™ environment

In this chapter, the development of a Windows ™ application to play 3-D sound on a PC in real time is explained, using double buffering to maintain a continuous flow of sound data. The only requirements for the developed application are a computer running Windows™ (Windows 95 and Windows 98 were tested) and a 16 bits sound board. Windows™ support different types of multimedia data, for example MIDI, waveform audio and video format. Waveform audio data is the type of interest in this thesis. For the double buffering, we basically set up two buffers and the system reads data using one buffer at a time. While one is being played, the other one is getting filled with 3-D sound. Internally, Windows™ keeps a linked list of the two buffers. Whenever one of these buffers becomes empty, it switches to the next buffer and calls a callback function to refill the buffer that has been emptied. The system then creates a new thread for managing the sound playback. Since a callback function is called only when the buffer is empty, the first buffer needs to be output before the main loop starts.

Audio playback is performed using a waveform audio output device (virtual device) or simply, a waveOut device. This device is available from the Win32 Multimedia API Reference. The Microsoft Multimedia API (Application Programming Interface) offers multimedia audio services through multimedia audio functions to control different types of waveform audio devices. These services allow the programmer to:

- Allocate, prepare and manage audio data blocks.
- Open and close audio device drivers.
- Query audio devices (for their capabilities) and handle errors with multimedia audio functions.

Real time programming needs some kind of parallel processing to be computationally feasible. Windows™ 95/98 is a single processor operating system and thus can only support parallelism through interleaving of processes and threads. Our application resides entirely on one PC and therefore we need the transfer of audio buffers to be executed as fast as possible to avoid interruption in audio playing. We used a shared-data model to perform inter-task communication. A shared-data model favors multi-threaded programming over multi-process programming, since tasks communicate using global variables residing in memory shared by all the tasks. It also prevents copying large amount of data relentlessly.

4.2 System architecture

Figure 4.1 shows the inter-task communication scheme in the single process program that was developed, containing two threads, which are called Main thread and WinPlayer thread.

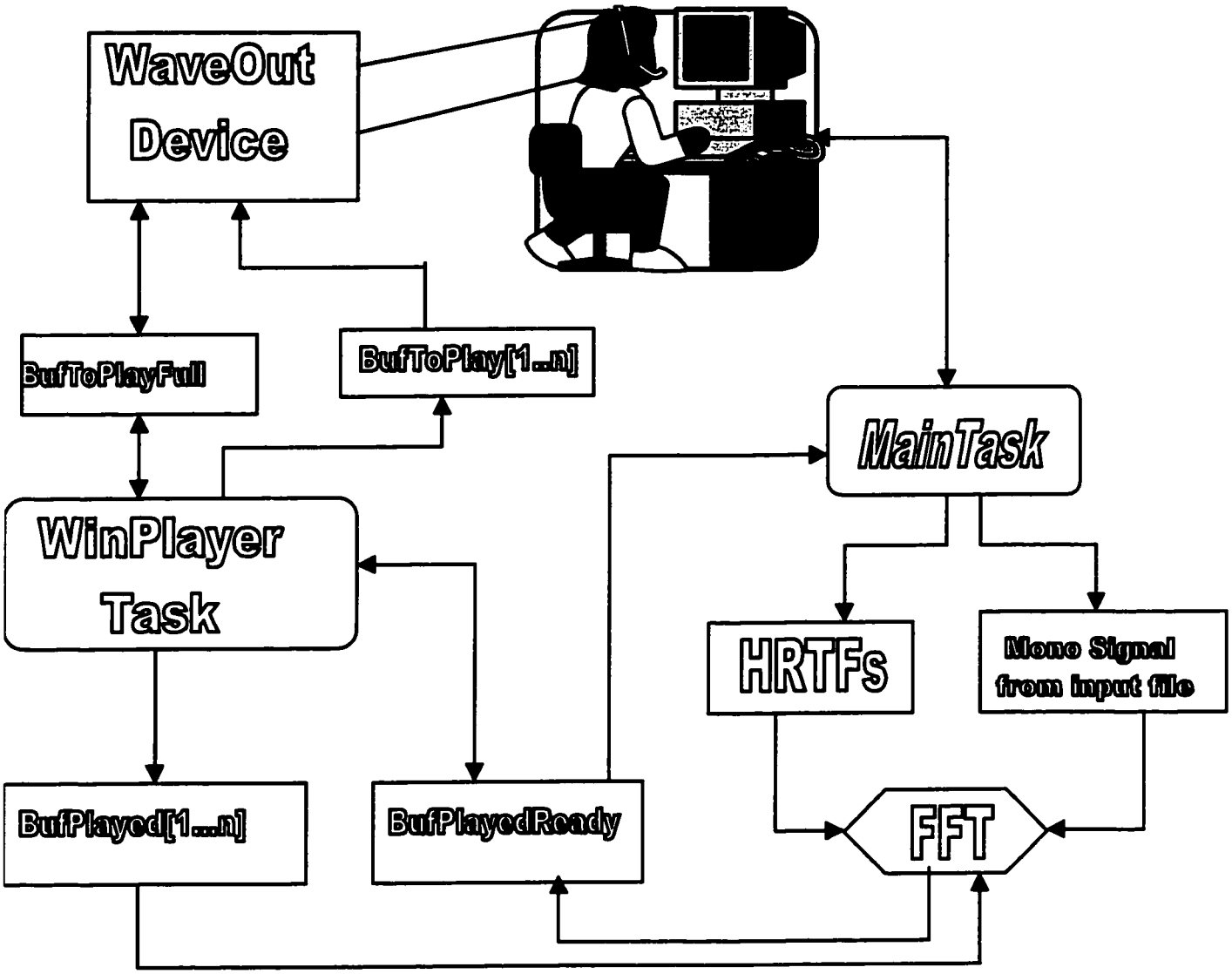

Figure 4.1. Inter-task communication scheme for the system.

The Main thread creates and terminates the WinPlayer thread, and retrieves the user input (in this case parameters such as source positions and time durations from another application). The decision states of the Main thread are shown in Figure 4.2. The Main thread is also responsible for allocating and initializing the global variables of the application, and shutting down the application. It opens the input WAVE file and gets the WAVE header. It chooses and retrieves the left and right impulse responses (HRIRs) for

a given location. Finally, it filters the input wave file with the HRIRs, to produce 3-D sound samples.

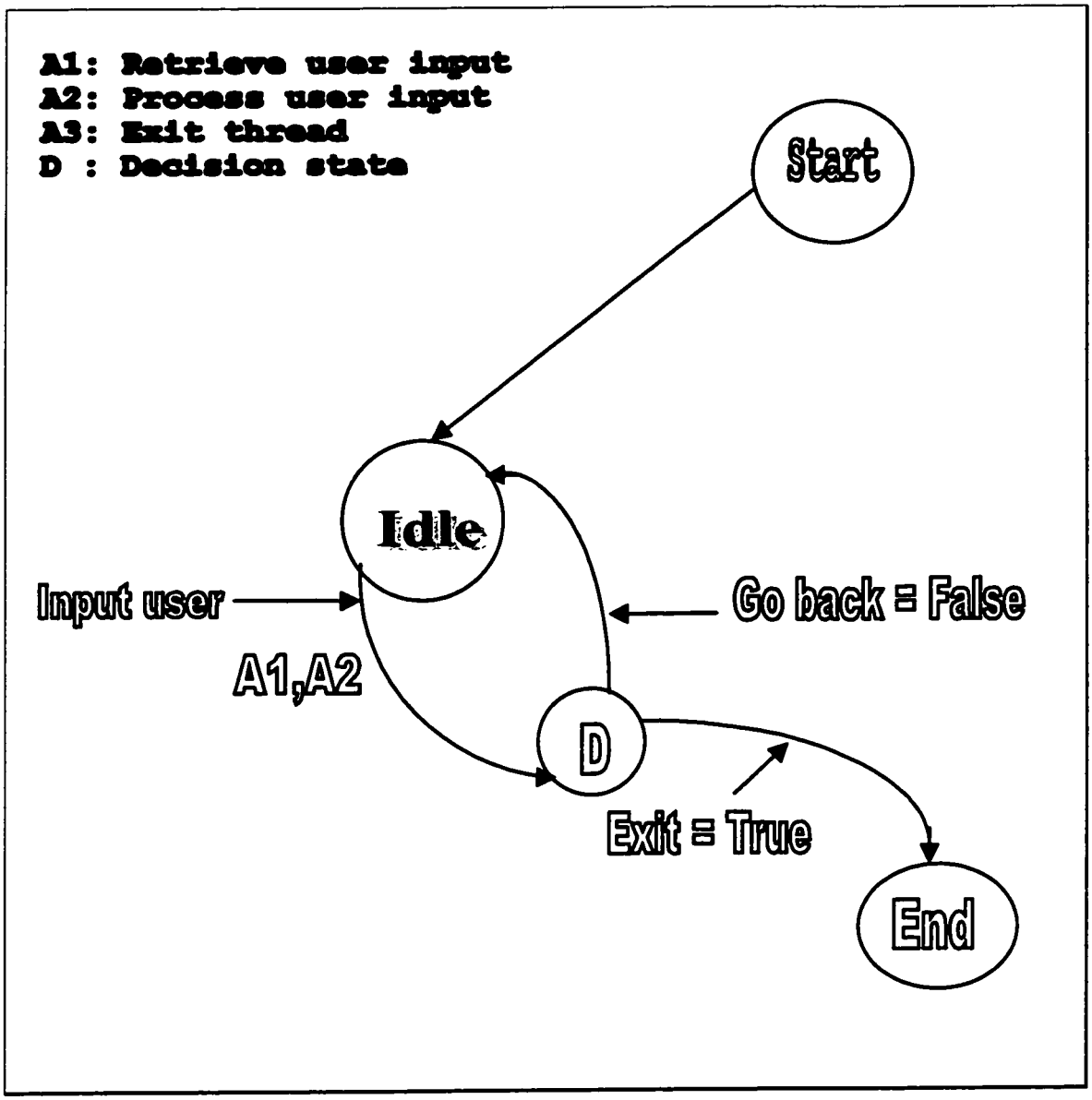

Figure 4.2. State Diagram of the Main Program thread.

The WinPlayer thread creates two buffers: one for playing an audio wave and another one for being refilled with data. Whenever the WaveOut device (see Figure 4.1) finishes playing a buffer, it requests a new one from the WinPlayer task, and immediately plays

the other buffer previously generated by the WinPlayer task. This technique is known as double buffering. The decision states of the WinPlayer task are shown in Figure 4.3. The WinPlayer task is responsible for generating audio samples to the BufToPlay buffer (see Figure 4.1) when the WaveOut device has finished to play a buffer (then BufToPlayFull is set to be FALSE). The WinPlayer task uses a "paused" variable to resume playback after the buffer is refilled, in case where the program is unable to provide data quickly enough.

It is the WinPlayer task that selects an audio device. Windows supports any number of audio devices, and selecting the right one can be tricky [32]. One way to find out the capabilities of the device is to request the device capabilities (using the waveOutGetDevCaps function of the Win32 Multimedia API). The other way is by calling the MinMaxChannels and MinMaxSamplingRate functions to determine the range of parameters acceptable to audio objects. Figure 4.3 shows that when the BufPlayedReady flag becomes FALSE, the WinPlayer task fills the next buffer to be played into BufPlayed and sets the BufPlayedReady flag to TRUE, to notify the Main task that a new buffer is ready to be played.

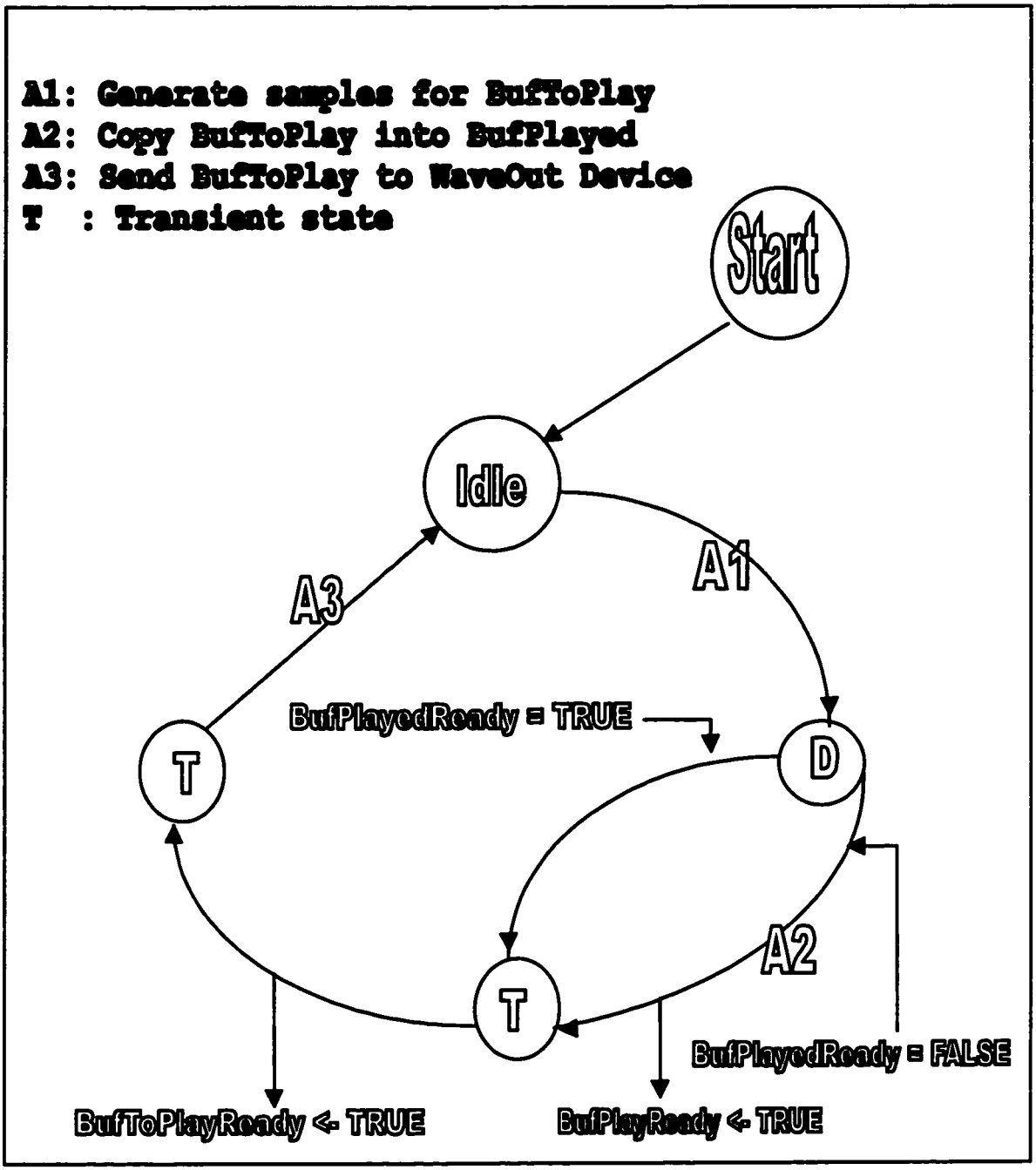

Figure 4.3. State Diagram of the WinPlayer thread.

4.3 First real-time implementation: convolution in the time domain

With the real-time structure described in the previous section, it was now possible to integrate the 3-D sound algorithm described by the time domain convolution equations (3.1) and (3.2). Note that equations (3.4)-(3.11) for the computation of the early reflections were dropped in the real-time version, because of the length of the "equivalent" HRIRs in this case, as explained by equation (3.12) and by an example in Section 3.2. Most of the MIT HRIRs sets were sampled at 44.1 kHz, except a "diffuse32k" set which was transformed from 44.1 kHz to 32 kHz. Since the computational load of operating in real-time at 32 kHz is obviously lower than operating at 44.1 kHz, it was chosen to use this HRIR set for the real-time implementation. Let's recall that in the offline version of Chapter 3 we had found very little difference (if any) in our listening tests with the "compact" and "diffuse" HRIR sets, so a similar performance can be expected by using "diffuse32k".

Even with the simplification of using only equations (3.1) and (3.2), the time domain convolution with the HRIRs produced some discontinuities in the output sound at 32kHz, because the application was not always capable of filling an output sound buffer while another buffer was played. This is clearly unacceptable for a real-time 3-D sound application. The maximum sampling rate that could be achieved without discontinuities with this time domain convolution method was 16kHz. Note that this is of course PC dependent, and a Pentium™ 200 MHz PC with a standard PCI 16 bits sound card was used for those measurements.

To obtain an HRIR set that could work at 16 kHz, it is possible to down-sample (i.e. low-pass filter and decimate) [31] the HRIRs, but this would obviously reduce the bandwidth of the frequency response of the HRIRs to less than 8 kHz, which may eliminate some important cues in frequencies higher than 8 kHz. As an alternative, it was decided to try to modify equations (3.1) and (3.2) in order to get exactly the same output samples but with a reduced complexity. This is described in the next section.

4.4 Second real-time implementation: improvement of the performance using frequency domain convolution

If the HRIRs and the input wave file signal are transformed into the frequency domain using the Fast Fourier Transform (FFT), they can simply be multiplied together and the result inversely transformed (with an IFFT) to obtain the same spatialized time domain output sounds of equations (3.1) and (3.2). Even with the additional computations involved in performing the FFTs on both the HRIR and the input wave signal and performing the inverse FFT of their product, the total computational load can be greatly reduced compared to the time domain convolution method. Due to some properties of the FFT, however, there are some slight complications when using the FFT for the linear convolution of two signals. The FFT method assumes that the signals are all periodic with the period being the length of the FFT size. Thus multiplication operations of the FFT of signals are analogous to circular convolution operations on their time domain counterparts [34].

In order to perform a linear convolution (and not a circular convolution) using the FFT of signals, a technique called "overlap and save" can be used [34]. This method will eliminate the 'wrap-around' effects of the circular convolution. The following steps describe this technique:

Definitions:

h[n]: an hrir filter (left or right),

M: length of the filter h[n],

L: Number of new input signal samples read in every block of samples, or number of valid output signal samples generated in every block of samples,

N = M+L-1 FFT size. Note that N has to be a power of 2 so that the FFT can be computed efficiently, and

x[n] and y[n]: input and output signals, respectively.

- Add L-1 zeros to h(n), then compute the FFT of h(n) to obtain H(k).
- Read L new samples of x[n].
- Combine the M-1 last samples of x[n] from the previous block with the L new samples of X[n], then apply the FFT to the resulting x[n] to obtain X(k).
- Y[k] = X[k] H[k].
- y[n]=IDFT(Y[k])
- the last L values of y[n] are valid outputs.

Figure 4.4 shows the alignment of the input and output signals for every block in the overlap and save technique:

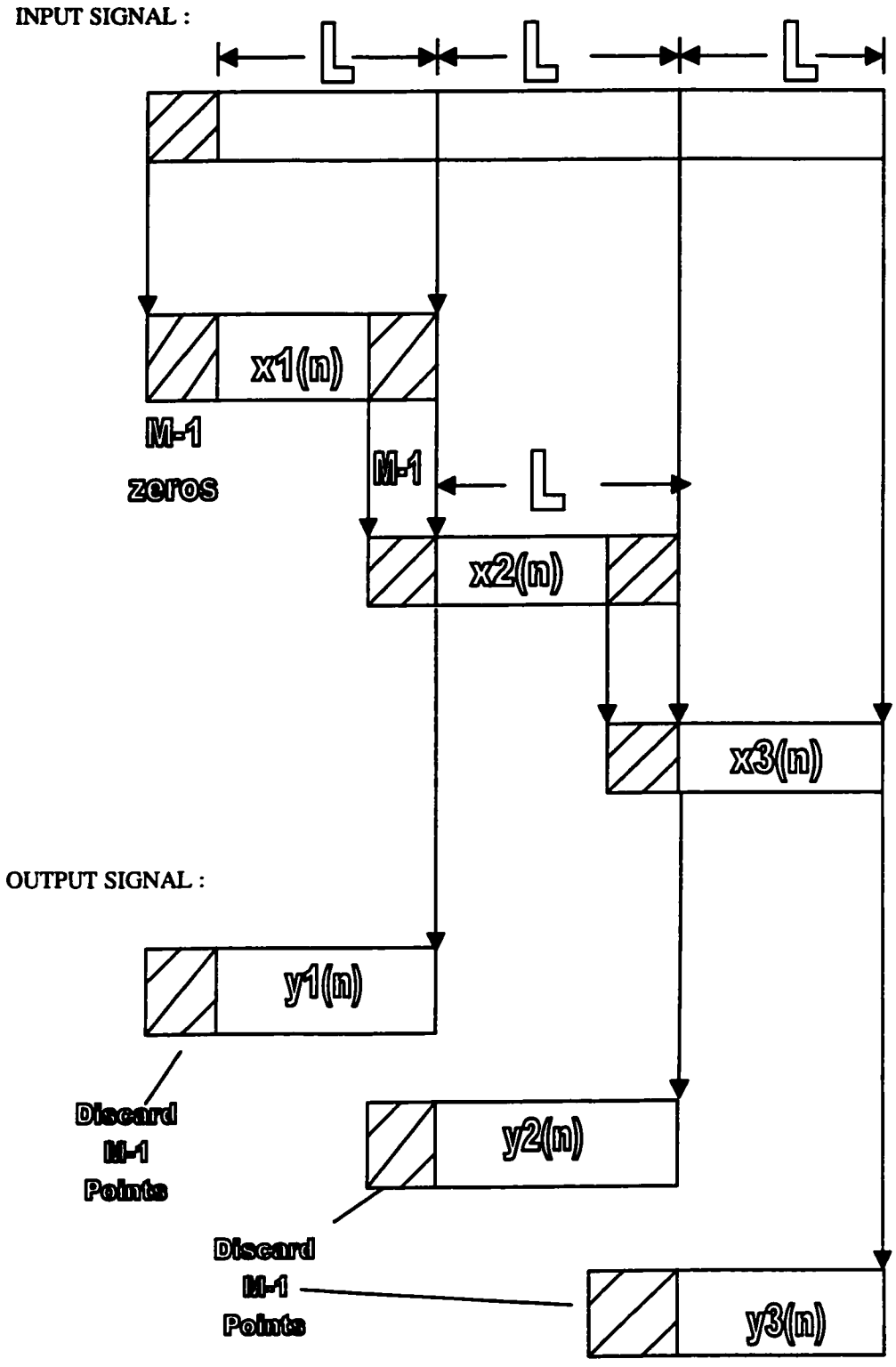

Figure 4.4. Alignment of input and output signals in the overlap and save technique.

In the frequency domain, using FFT with the overlap-save technique, the computational cost per output sample is reduced. The number of real multiplies required per output sample is $2 \times M$ for the time domain linear convolution and $4 \times 2 \times \frac{N \log_2 2N}{L}$ for the frequency domain linear convolution using overlap and save. The factor 2 is because two channels are computed, and the factor 4 for the frequency domain technique is because any complex multiply requires 4 real multiplies. To illustrate this further, Table 4.1 compares for different values of M and L the number of real multiplies per output sample required with the two methods.

M	L	N=L+M-1	Multiplies/sample for Time domain Linear. Convolution	Multiplies/sample for Freq. Domain Linear Convolution using overlap and save
32	225	256	64	82.7
128	129	256	256	145.1
32	993	1024	64	90.9
128	897	1024	256	100.7
512	513	1024	1 024	88.0
32	16 353	16 384	64	120.2
128	16 257	16 384	256	121.0
512	15 873	16 384	1 024	123.9
32	1 048 545	1 048 576	64	168.0
128	1 048 449	1 048 576	256	168.0
512	1 048 065	1 048 576	1 024	168.1

Table 4.1. Computational cost per output sample for time domain and freq. domain linear convolution techniques.

From Table 4.1, a few conclusions can be found. First of all, for low values of M (such as 32) the time domain linear convolution method requires less computations and should therefore be used. For higher values of M (such as 128 and 512) the frequency domain

linear convolution method using overlap and save technique requires less computations and should be used. For a given value of M, there is an optimum value of N (not too low, not too high) for which the number of multiplies is minimized. From Table 4.1, the optimum for M=128 appears to be near N=1 024. In our application, the HRIRs also had a length M of 128 samples, but a higher value than 1 024 was chosen for N. This is because of the additional constraint that some overhead is introduced in the real-time application every time a switch occurs in the double buffering process. N=16 384 was selected instead, and therefore the data block size L was 16 257 samples.

With the overlap and save frequency domain technique using M=128, L=16 257 and N=16 384, the maximum sampling rate frequency that could be achieved to play 3-D sound audio samples in an uninterrupted fashion was 32kHz, without any discontinuity in the output sound. This is twice as high as the maximum sampling rate that was achieved with the time domain convolution technique, as reported in the previous section. As for the numerical results obtain with the two techniques (when there is no discontinuity in the output), it is a well known fact that they both produce the same result since they are both computing a linear convolution, and this was verified at the debugging stage of the project.

It should be noted that the frequency domain technique would also be directly applicable for the implementation of a 3-D sound system with reverberation, as the one described in Section 3.2. The only difference would be that the equivalent HRIRs as computed by equations (3.10) and (3.11) would be used instead of the anechoic HRIRs. The reduction

of computational complexity would be even greater than for anechoic HRIRs, since the equivalent HRIRs typically have a much greater length, and the frequency domain technique is particularly well suited for filters with long lengths, as shown by table 4.1.

The complete source code for the real-time 3-D sound rendering system using the frequency domain technique for the linear convolution is provided in the Appendices. First, a "readme" text file describing the different source code files is shown in Appendix A. Appendix B lists all the C++ source code for the real-time 3-D application, to be compiled as a DLL. Appendix C shows the C++ source code of a simple application (to be compiled as an executable) that calls the 3-D audio DLL for a specific listener position, listener direction and sound source position.

With a real-time application now running with no discontinuity at 32 kHz, it was then possible to use properly the "diffuse32k" MIT HRIR set, and it was possible to perform some listening tests to validate the performance of the system, as explained in the next section.

4.5 Listening tests for the real-time 3-D sound application

As expected, the listening tests performed with the real-time implementation of the 3-D sound software produced the same subjective results as those reported in section 3.5 for the offline 3-D audio system (for the special case where no reflections were included in the offline system). This comes as no surprise since the "diffuse32" and "diffuse" MIT HRIR sets are basically the same up to a bandwidth of 16 kHz. Once again for a sound

source moving around the listener's head, the listening results were quite realistic, at least for the candidate and his supervisor. As explained in Section 3.5, no detailed subjective listening tests were performed, because this was obviously not a main objective of the project, and it is possible at any time to replace a set of HRIR by another one if it performs better.

4.6 Chapter summary

In this chapter, details of the real-time implementation of a 3-D sound generation system using binaural technology were presented. The first two sections provided details of the multi-threads approach that was used. The next sections described the time domain and frequency domain techniques that were used to perform the HRIR filtering of a mono sound source, and the performance of each technique was experimentally measured with the maximum sampling rate that was achieved with each technique. Finally, it was verified that the subjective listening performance of the real-time 3-D sound system was the same as the offline 3-D sound system described in Chapter 3, for the particular case where no reflections were used.

Chapter 5 Conclusion and Future Work

5.1 Conclusion

In this thesis, we used the technique of binaural 3-D audio to develop a real-time implementation of 3-D sound software. The theory of sound spatialization was first reviewed so that it would be clear which 3-D effects could be reproduced by the system. Then an intermediate offline version was developed, so that the algorithms could be tested without the real-time constraint. In the offline version it was also possible to add some effects such as early reflections in a virtual rectangular room. Once the offline system performed satisfactory it was possible to adapt it to a real-time implementation (but without the early reflections). The initial real-time implementation used time-domain convolutions to perform the HRIR filtering. It was found however that an improvement using frequency domain linear convolution with an overlap and save technique was required, in order to achieve a sampling rate sufficiently high so that the application could be directly used with a MIT HRIR set.

As mentioned in this thesis, the conditions of the binaural system that we used were far from the ideal individualized HRIRs and individualized compensation (in fact there was no compensation at all). Nevertheless, the spatialization performance of the 3-D sound system appeared to be good in some preliminary listening tests accomplished by the candidate and his supervisor. In any case, the developed software could easily be adapted to use other sets of HRIRs, individualized or not, including compensation or not.

5.2 Future Works

Extensions to this thesis would be to:

1. Perform more extensive and controlled psychoacoustic listening tests to evaluate the spatialization capabilities of a given set of HRIRs,
2. Use a head-tracking system on the listener so that the virtual environment would not move with the head of a listener,
3. Synchronize the 3-D sound rendering with images and other modalities in a virtual environment,
4. Use loudspeakers with crosstalk cancellation instead of headphones, and
5. Add compensation (adaptive or not) to cancel the headphone effect during the playback.

Also, the investigation and testing of recently introduced software tools that can generate 3-D sounds for virtual environments such as the Direct Sound™ interface [35] or Java 3-D audio [36] would be interesting.

References

[1] Begault, D. R. (1994). "3-D Sound for Virtual Reality and Multimedia Applications". Academic Press, Cambridge, MA.

[2] Kendall, G. S. (1995). "3-D sound primer: directional hearing and stereo reproduction", *Computer Music Journal.* v 19 n 4. p 23-46.

[3] Moller, H. (1992). "Fundamentals of binaural technology.", *Applied Acoustics.* v 36 n 3-4 p 171-218.

[4] Bauck, J., and Cooper, D. H. (1996). "Generalized transaural stereo and applications", *Journal of the Audio Engineering Society.* v 44 n 9 p 683-705.

[5] Malham, D. G. and Myatt, A. (1995). "3-D sound spatialization using ambisonic techniques", *Computer Music Journal.* v 19 n 4. p 58-70.

[6] McEwan, J A . (1994). "Real-time 3-D: the sound of the times". *GEC Review.* v 9 n 2 1994. p 67-80.

[7] Gardner, W. G. (1997). "Head Tracked 3-D Audio Using Loudspeakers", *Proc. IEEE* Workshop on Applications of Signal Processing to Audio and Acoustics, New Paltz, NY.

[8] Berg, R. E. and Stork D. G. (1982). "The Physics of Sound". Prentice-Hall, Englewood Cliffs, NJ.

[9] Dempsey, M.J. (1997). "How Positional 3-D Audio Works"; VLSI Technology.inc. *White Paper* http://www.vlsi.com .

[10] Sibbald, A. (1994). "An introduction to sound and hearing. ". *"http://www.sensaura.co.uk/wse/Tech%20Page/Devpc005.pdf".*

[11] Rigden, J. S.(1985). "Physics and the Sound of Music" J. Wiley 2nd ed.

[12] Sibbald, A. (1995). "An introduction to Digital Earä technology". *White Paper http://www.sensaura.co.uk/wse/Tech%20Page/Devpc003.pdf*

[13] http://www.headwize.com/tech/aureal1_tech.htm "3-D Audio Primer " *Aureal Corporation.*

[14] Sibbald, A. (1996). "Virtual audio for headphones". *Headwize Technical Papers (http://www.headwize.com/tech/sibbald_tech.htm)*

[15] Wightman ,F L and Kistler D. J. (1989). "Headphone simulation of free-field listening. 2: Psychophysical validation." *Acoust. Soc.* Am., 85, (2), pp. 868-878.

[16] Blauert, J. (1997). "Spatial Hearing" 2nd. *MIT Press*, Cambridge, MA.

[17] Sibbald, A. (1994). "Virtual Earä technology". *http://www.sensaura.co.uk/wse/Tech%20Page/Devpc011.pdf.*

[18] Handel, S. (1989). "A recommended introductory textbook on the psychology of hearing; includes non-mathematical chapters on the physics of sound production, propagation and diffraction, plus a lucid chapter on auditory neurophysiology". *Listening (MIT Press, Cambridge, MA).*

[19] Duda, R. O. (1996). "3-D Audio for HCI," in *Proc. Twenty-Seventh Annual Asilomar Conference on Signals, Systems and Computers* (Asilomar, CA).

[20] Brown, C. P. (1996). "Modeling the Elevation Characteristics of the Head-Related Impulse Response," *Technical Report* No. 13, NSF Grant No. IRI-9402246, Dept. of Elec. Engr., San Jose State Univ., San Jose, CA.

[21] Han, H. L. (1994). "Measuring a dummy head in search of pinna cues", *J. Audio Eng. Soc.*, Vol. 42, No. 1/2, pp. 15-37

[22] Raghunath Rao, K. and Jezekiel Ben-Arie (1996). "Optimal Head Related Transfer Functions for Hearing and Monaural Localization in Elevation: A Signal Processing

Design Perspective", *IEEE Trans. On Biomed. Eng.* Vol. 43, No. 11, Nov. pp. 1093 –1104.

[23] Duda, R. O. (1993). "Modeling head related transfer functions," in *Proc. Twenty-Seventh Annual Asilomar conference on Signals, Systems and Computers,* Asilomar, CA.

[24] Gardner, W.G., and Martin, K.D.(1995). "HRTF measurements of a KEMAR , *J. Audio Eng. Soc.*, 43(3), pp. 3907-3908.

[25] Wenzel, E. M., Arruda, M. Kistler, D. J. and Wightman, F. L. (1993). "Localization using nonindividualized head-related transfer functions", *J. Acoust. Soc. Am.*, 94(1), pp. 111-123.

[26] Gardner, W.G. (1998). "3-D Audio and Acoustic Environment Modeling", Kluwer Academic, Norwell, MA.

[27] Gardner, W. G. (1995). "Transaural 3-D audio", *M.I.T. Media Lab* Perceptual Computing Section. *http://sound.media.mit.edu/papers.html.*

[28] Sibbald, A.(1996) "Transaural acoustic crosstalk cancellation". *"http://www.sensaura.co.uk/wse/Tech%20Page/Devpc009.pdf".*

[29] Zudock, T. (1996) "Virtual audio through ray tracing", *Dr Dobb's Journal - Software Tools for the Professional Programmer,* v.21, n.12, p.34, 12p.

[30] Hammershoi, D and Sandvad, J. (1994). "Binaural auralization. Simulating free field conditions by headphones". *Proc. AES, 96th Convention,* Amsterdam.

[31] Gilkey, R. H., and Anderson, T. R. Ed (1997). Binaural and Spatial Hearing in Real and Virtual Environments, Lawrence Erlbaum Associates, Mahwah, NJ.

[32] Kientzle, T. (1998). "A Programmer's Guide to Sound". *Academic Press,* Cambridge, MA.

[33] *http://www.univ-lyon1.fr/~jd/groovit/analog/wave/wave.pdf* "wave file format definition" Groovit.

[34] Proakis, J. D. and Manolakis, D. G. (1992). "Digital Signal Processing". *Principles, Algorithms, and Applications"*, 2nd Ed. Macmillan, NY.

[35] Jorg, K. "Direct Sound (DirectX 5.0)," *http://www.ews64.com/mcdirectsound.html*

[36] Henry A. S., and Michael F. D (1999) "The Java 3D API and Virtual Reality," *IEEE Computer Graphics and Applications*, May/June, 1999

Appendix A "readme.txt" file

Readme file for real time 3-D audio system application

Written By: Adel Senki (1646856)

Date: Jan 15, 2000

It is a single-process program that contain two threads:

User Interface thread:

- def.h // general application header file.

- prp.cpp // main application file.

WinPlayer thread:

- audio.h // header file for base class audioAbstract

- audio.h // C++ file for audio.h

- audiosource.h // header file for class DynamicAudioSource.

- audiosource.cpp // C++ file for AudioSource.h

- winplayer.h // header file for class WinPlayer, which is derived from the
// DynamicAudioSource class

- winplayer.cpp // C++ file for winplayer.h

Appendix B C++ source code for the 3-D sound application (to be compiled as a DLL)

DEF.H

```
/*************************************************************
/* File Name : def.h
/* Purpose : Main application header file
/* Written by : Adel Senki (#1646856)
/* Date : Feb 5 1999
/*************************************************************/
#ifndef DEF_H_INCLUDED
#define DEF_H_INCLUDED
#ifndef winBufferSize
#define winBufferSize 16257 // Number of samples per buffer of audio
#define samsize 16257
#endif
// Types
typedef short AudioSample;
typedef unsigned char AudioByte;
// External Variables
extern          float     *Samples;
extern          long              Samples_BufSize;      // input binary wave file
extern          short     Sample[samsize];               // variable for reading from file
extern          unsigned long TotalElapsedSamps;       // elapsed time since beginning
extern          unsigned long TotalSamplesToRead;      // fro  the source WAV file
extern          float             *left_reverb_HRTF;
extern          float             *right_reverb_HRTF;
extern          float             *im_l_reverb_HRTF;
extern          float             *out_l_reverb_HRTF;
extern          float             *imgout_l_reverb_HRTF;
extern          float             *im_r_reverb_HRTF;
extern          float             *out_r_reverb_HRTF;
extern          float             *imgout_r_reverb_HRTF;
extern          float             *im_input ;
extern          float             *out_Sample;
extern          float             *imgout_Sample;
extern          unsigned          Total_sam;
extern          float             *out_L_Sample;
extern          float             *imgout_L_Sample;
extern          float             *out_R_Sample;
extern          float             *imgout_R_Sample;
extern          float             *Output_left;          // the value of the current output
extern          float             *Output_right;
extern          float                     *imgout_left;          // the value of the current output
extern          float             *imgout_right;
#endif
```

PRP.CPP (Main Program)

```
/*****************************************************************************/
/* Purpose : Main program file. The main program is responsible for allocating and initializing the global
/* variables of the  application, managing all threads and shutting down the application.
/* Written by : Adel Senki (1646856)
/*****************************************************************************/

#INCLUDE "DEF.H"
#include "fourier.h"
// Header files for playing audio
#include "audio.h"
#include "winplayr.h"
#include "audiosource.h"
// System header files
#include <stdlib.h>
#include <iostream.h>
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <math.h>
// macro defined constants
#define Vs 334.0
#define PI 3.14159265359
#define REFDIST 1.0
#define HRTF_LEN 128
extern   FILE    *InFileBin;                    // input binary wave file
// structure for a point in 3 dimensional space
struct point
{        float X;
float Y;
        float Z;      };
// structure for a point paired with a time duration
struct position
{  struct point Coord;
        float Time;   };
// function prototypes
void CalcImpResp(float *, float *,struct point , struct point, int, struct point, char *, char *);
long OpenWaveFile( FILE *, PCMWAVEFORMAT *);
void InputRiffId(char *,FILE *);
void AddWaveHeader( FILE *, PCMWAVEFORMAT);
// structure used to pass information between threads
typedef struct tagThreadInfo {
long      samplingRate;
        int       channels;
  } ThreadInfo;
// Thread handles
HANDLE g_hPlayThread;
DWORD dwPlayThreadID;
DWORD WINAPI PlayThread(LPVOID lpParm);
extern "C" char _export   playsound(char *hrtf_path,char *hrtf_sub_path, char *InFileName,struct point
Listener_orientation, struct point Listener,struct position CurrentSource)
{  PCMWAVEFORMAT WaveHeader;                    // structure to hold wave header vals
  float           Fs;
  int             i;                            // the sampling frequency
```

```
// open input WAVE file
        if ((InFileBin=fopen(InFileName,"rb"))==NULL)
  {     MessageBox(NULL,"Unable to open input wave file!",
                "ERROR", MB_OK|MB_ICONEXCLAMATION);
  exit(-1) ;      }
  Total_sam=Samples_BufSize+ samsize-1 ;
// Get WAVE header, use input header info to set Fs
        OpenWaveFile(InFileBin,&WaveHeader);
        Fs = WaveHeader.wf.nSamplesPerSec;
// Get the source positions and durations
  TotalSamplesToRead +=(CurrentSource.Time)*Fs;
  Samples_BufSize= HRTF_LEN;
  if((Samples =(float *) malloc(sizeof(float)*Total_sam))==NULL)
        {       MessageBox(NULL,"Error: Unable to allocate memory1","message",
                                        MB_OK|MB_ICONEXCLAMATION);
                exit(-1);       }
  if((im_input =(float *) malloc(sizeof(float)*Total_sam))==NULL)
        {       MessageBox(NULL,"Error: Unable to allocate memory2","message",
                                        MB_OK|MB_ICONEXCLAMATION);
                exit(-1);       }
  if((out_Sample =(float *) malloc(sizeof(float)*Total_sam))==NULL)
        {       MessageBox(NULL,"Error: Unable to allocate memory3","message",
                                        MB_OK|MB_ICONEXCLAMATION);
                exit(-1);       }
       if((imgout_Sample =(float *) malloc(sizeof(float)*Total_sam))==NULL)
        {       MessageBox(NULL,"Error: Unable to allocate memory4","message",
                                        MB_OK|MB_ICONEXCLAMATION);
                exit(-1);       }
        if((left_reverb_HRTF =(float *) malloc(sizeof(float)*Total_sam))==NULL)
        {       MessageBox(NULL,"Error: Unable to allocate memory5","message",
                                        MB_OK|MB_ICONEXCLAMATION);
                exit(-1);       }
        if((right_reverb_HRTF =(float *) malloc(sizeof(float)*Total_sam))==NULL)
        {       MessageBox(NULL,"Error: Unable to allocate memory6","message",
                                        MB_OK|MB_ICONEXCLAMATION);
                exit(-1);       }
  if((im_l_reverb_HRTF =(float *) malloc(sizeof(float)*Total_sam))==NULL)
        {       MessageBox(NULL,"Error: Unable to allocate memory7","message",
                                        MB_OK|MB_ICONEXCLAMATION);
                exit(-1);       }
  if((out_l_reverb_HRTF =(float *) malloc(sizeof(float)*Total_sam))==NULL)
        {       MessageBox(NULL,"Error: Unable to allocate memory","message",
                                        MB_OK|MB_ICONEXCLAMATION);
                exit(-1);       }
       if((imgout_l_reverb_HRTF =(float *) malloc(sizeof(float)*Total_sam))==NULL)
        {       MessageBox(NULL,"Error: Unable to allocate memory8","message",
                                        MB_OK|MB_ICONEXCLAMATION);
                exit(-1);       }
  if((im_r_reverb_HRTF =(float *) malloc(sizeof(float)*Total_sam))==NULL)
        {       MessageBox(NULL,"Error: Unable to allocate memory9","message",
                                        MB_OK|MB_ICONEXCLAMATION);
                exit(-1);       }
  if((out_r_reverb_HRTF =(float *) malloc(sizeof(float)*Total_sam))==NULL)
        {       MessageBox(NULL,"Error: Unable to allocate memory","message",
                                        MB_OK|MB_ICONEXCLAMATION);
                exit(-1);       }
```

```
        if((imgout_r_reverb_HRTF =(float *) malloc(sizeof(float)*Total_sam))==NULL)
        {       MessageBox(NULL,"Error: Unable to allocate memory","message",
                                        MB_OK|MB_ICONEXCLAMATION);
                        exit(-1);       }
if((imgout_L_Sample =(float *) malloc(sizeof(float)*Total_sam))==NULL)
        {       MessageBox(NULL,"Error: Unable to allocate memory","message",
                                        MB_OK|MB_ICONEXCLAMATION);
                        exit(-1);       }
if((out_L_Sample =(float *) malloc(sizeof(float)*Total_sam))==NULL)
        {       MessageBox(NULL,"Error: Unable to allocate memory","message",
                                        MB_OK|MB_ICONEXCLAMATION);
                        exit(-1);       }
if((Output_left =(float *) malloc(sizeof(float)*Total_sam))==NULL)
        {       MessageBox(NULL,"Error: Unable to allocate memory","message",
                                        MB_OK|MB_ICONEXCLAMATION);
                        exit(-1);       }
        if((imgout_left =(float *) malloc(sizeof(float)*Total_sam))==NULL)
        {       MessageBox(NULL,"Error: Unable to allocate memory","message",
                                        MB_OK|MB_ICONEXCLAMATION);
                        exit(-1);       }
if((imgout_R_Sample =(float *) malloc(sizeof(float)*Total_sam))==NULL)
        {       MessageBox(NULL,"Error: Unable to allocate memory","message",
                                        MB_OK|MB_ICONEXCLAMATION);
                        exit(-1);       }
if((out_R_Sample =(float *) malloc(sizeof(float)*Total_sam))==NULL)
        {       MessageBox(NULL,"Error: Unable to allocate memory","message",
                                        MB_OK|MB_ICONEXCLAMATION);
                        exit(-1);       }
if((Output_right =(float *) malloc(sizeof(float)*Total_sam))==NULL)
        {       MessageBox(NULL,"Error: Unable to allocate memory","message",
                                        MB_OK|MB_ICONEXCLAMATION);
                        exit(-1);       }
        if((imgout_right =(float *) malloc(sizeof(float)*Total_sam))==NULL)
        {       MessageBox(NULL,"Error: Unable to allocate memory","message",
                                        MB_OK|MB_ICONEXCLAMATION);
                        exit(-1);       }
 for (i=0;i<Total_sam;i++)
        {       Samples[i]=0.0;
        im_input[i]=0.0;
                out_Sample[i]=0.0;
                imgout_Sample[i]=0.0;
                im_l_reverb_HRTF[i]=0.0;
                im_r_reverb_HRTF[i]=0.0;
                out_l_reverb_HRTF[i]=0.0;
                imgout_l_reverb_HRTF[i]=0.0;
                imgout_L_Sample[i]=0.0;
                imgout_r_reverb_HRTF[i]=0.0;
                imgout_L_Sample[i]=0.0;
                out_L_Sample[i]=0.0;
                Output_left[i]=0.0;
                imgout_left[i]=0.0;
                imgout_R_Sample[i]=0.0;
                out_R_Sample[i]=0.0;
                Output_right[i]=0.0;
                 imgout_right[i]=0.0;
        }
```

```
// calculate the response for the first position
CalcImpResp(left_reverb_HRTF,right_reverb_HRTF,(CurrentSource.Coord), Listener,
Samples_BufSize,Listener_orientation,hrtf_path,hrtf_sub_path);
fft_float(Total_sam,0,left_reverb_HRTF,im_l_reverb_HRTF,out_l_reverb_HRTF,imgout_l_reverb_HRTF)
;
fft_float(Total_sam,0,right_reverb_HRTF,im_r_reverb_HRTF,out_r_reverb_HRTF,imgout_r_reverb_HRT
F);
  ThreadInfo threadInfo;
  threadInfo.samplingRate=22050;
  threadInfo.channels=2;
  //Play
  // Create thread for playing */
  g_hPlayThread = CreateThread(NULL, NULL, PlayThread, (LPVOID)&threadInfo,
                    0, &dwPlayThreadID);
  if (g_hPlayThread == NULL) {
    cerr << "prp.cpp: Error! creating play thread.";
    cerr << "GetLastError(): " << GetLastError() << ".\n";
  }
   if (!SetThreadPriority(g_hPlayThread, THREAD_PRIORITY_HIGHEST                    )) {
    cerr << "prp.cpp: Error! setting play thread priority.\n";
    cerr << "GetLastError(): " << GetLastError() << ".\n";
  }
  Sleep(500 /* ms */);
  // Resume suspend play and record threads
  while ( ResumeThread(g_hPlayThread) == 1 ) {
    Sleep(5 /* ms */);
    cerr << "Waiting for play thread to be suspended.\n";
  }
  cerr << "Resumed play thread.\n";
   while (1)
 {    {
          Sleep(5 /* ms */);
    }  }
fclose(InFileBin);
free(Samples);
free(im_input);
free(out_Sample);
free(imgout_Sample);
free(left_reverb_HRTF);
free(right_reverb_HRTF);
free(im_l_reverb_HRTF);
free(out_l_reverb_HRTF);
free(imgout_l_reverb_HRTF);
free(im_r_reverb_HRTF);
free(out_r_reverb_HRTF);
free(imgout_r_reverb_HRTF);
free(imgout_L_Sample);
free(out_L_Sample);
free(Output_left);
free(imgout_left);
free(imgout_R_Sample);
free(Output_right);
free(imgout_right);
free(out_R_Sample);
return (0);
}
```

```
DWORD WINAPI PlayThread(LPVOID lpParm) {
  ThreadInfo * pThreadInfo = (ThreadInfo *) lpParm;

  WinPlayer player(pThreadInfo->samplingRate, pThreadInfo->channels);

  cerr << "suspending play thread.\n";
  SuspendThread(g_hPlayThread);
  cerr << "prp.cpp: Start playing.\n";
  player.Play();
  cerr << "prp.cpp: Finished playing audio source.\n";
  return (0);
}

/*****************************************************************************
CalcImpResp
This subroutine calculates the time delays (in samples) and
attenuations for the early reflections in the room.
*****************************************************************************/
void CalcImpResp(float *left_reverb_HRTF,float *right_reverb_HRTF,
struct point Source,struct point Listener, int Samples_BufSize,
      struct point Listener_orientation, char *hrtf_path, char *hrtf_sub_path)
{
          float Dist;                   // distance travelled by sound ray
          struct point MirrSource;      // mirrored source x,y,z coords
  float Gain;
          int i,l;
  short int int_temp;
  float theta, phi, r, azimuth, elevation;
  float float_temp;
  float elevation_values[14]={-40,-30,-20,-10,0,10,20,30,40,50,60,70,80,90};
  long  azimuth_nb_of_values[14]={56,60,72,72,72,72,72,60,56,45,36,24,12,1};
  float **azimuth_values_pointers;
  float azimuth_values_m40_elev[56]={0,6,13,19,26,32,39,45,51,58,
  64,71,77,84,90,96,103,109,116,122,129,135,141,148,154,161,167,174,180,186,
  193,199,206,212,219,225,231,238,244,251,257,264,270,276,283,289,296,302,309,315,

          321,328,334,341,347,354};
 float azimuth_values_m30_elev[60]={0,6,12,18,24,30,36,42,48,54,60,66,72,78,84,
  90,96,102,108,114,120,126,132,138,144,150,156,162,168,174,180,186,192,198,204,
  210,216,222,228,234,240,246,252,258,264,270,276,282,288,294,
                300,306,312,318,324,330,336,342,348,354};
 float azimuth_values_m20_elev[72]={0,5,10,15,20,25,30,35,40,45,50,55,60,65,70,
  75,80,85,90,95,100,105,110,115,120,125,130,135,140,145,150,155,160,165,170,
  175,180,185,190,195,200,205,210,215,220,225,230,235,240,245,250,255,260,265,
  270,275,280,285,290,295,300,305,310,315,320,325,330,335,340,345,350,355};
 float azimuth_values_m10_elev[72]={0,5,10,15,20,25,30,35,40,45,50,55,60,65,70,
 75,80,85,90,95,100,105,110,115,120,125,130,135,140,145,150,155,160,165,170,
 175,180,185,190,195,200,205,210,215,220,225,230,235,240,245,250,255,260,265,270
 ,275,280,285,290,295,300,305,310,315,320,325,330,335,340,345,350,355};
 float azimuth_values_0_elev[72]={0,5,10,15,20,25,30,35,40,45,50,55,60,65,70,
 75,80,85,90,95,100,105,110,115,120,125,130,135,140,145,150,155,160,165,170,175,
 180,185,190,195,200,205,210,215,220,225,230,235,240,245,250,255,260,265,270,275,
 280,285,290,295,300,305,310,315,320,325,330,335,340,345,350,355};
 float azimuth_values_10_elev[72]={0,5,10,15,20,25,30,35,40,45,50,55,60,65,70,
 75,80,85,90,95,100,105,110,115,120,125,130,135,140,145,150,155,160,165,170,175,
 180,185,190,195,200,205,210,215,220,225,230,235,240,245,250,255,260,265,270,
```

```
275,280,285,290,295,300,305,310,315,320,325,330,335,340,345,350,355};
float azimuth_values_20_elev[72]={0,5,10,15,20,25,30,35,40,45,50,55,60,65,70,
75,80,85,90,95,100,105,110,115,120,125,130,135,140,145,150,155,160,165,170,175,
180,185,190,195,200,205,210,215,220,225,230,235,240,245,250,255,260,265,270,
275,280,285,290,295,300,305,310,315,320,325,330,335,340,345,350,355};
float azimuth_values_30_elev[60]={0,6,12,18,24,30,36,42,48,54,60,66,72,78,84,
90,96,102,108,114,120,126,132,138,144,150,156,162,168,174,180,186,192,198,204,
210,216,222,228,234,240,246,252,258,264,270,276,282,288,294,
            300,306,312,318,324,330,336,342,348,354};
float azimuth_values_40_elev[56]={0,6,13,19,26,32,39,45,51,58,
64,71,77,84,90,96,103,109,116,122,129,135,141,148,154,161,167,174,180,186,
193,199,206,212,219,225,231,238,244,251,257,264,270,276,283,289,296,302,309,315,
                  321,328,334,341,347,354};
float azimuth_values_50_elev[45]={0,8,16,24,32,40,48,56,64,72,
80,88,96,104,112,120,128,136,144,152,160,168,176,184,192,200,208,216,224,232,
240,248,256,264,272,280,288,296,304,312,320,328,336,344,352};
float azimuth_values_60_elev[36]={0,10,20,30,40,50,60,70,80,90,
100,110,120,130,140,150,160,170,180,190,200,210,220,230,240,250,260,270,280,290,

                                    300,310,320,330,340,350};
float azimuth_values_70_elev[24]={0,15,30,45,60,75,90,105,120,135,
150,165,180,195,210,225,240,255,270,285,300,315,330,345};
float azimuth_values_80_elev[12]={0,30,60,90,120,150,180,210,240,270,300,330};
float azimuth_values_90_elev[1]={0};
char file_name[256];
char StrLine[400];
FILE *hrtf_file_pointer;
float file_left_hrtf[HRTF_LEN];
float file_right_hrtf[HRTF_LEN];

azimuth_values_pointers = (float **) malloc(14*sizeof(float *));
azimuth_values_pointers[0]=azimuth_values_m40_elev;
azimuth_values_pointers[1]=azimuth_values_m30_elev;
azimuth_values_pointers[2]=azimuth_values_m20_elev;
azimuth_values_pointers[3]=azimuth_values_m10_elev;
azimuth_values_pointers[4]=azimuth_values_0_elev;
azimuth_values_pointers[5]=azimuth_values_10_elev;
azimuth_values_pointers[6]=azimuth_values_20_elev;
azimuth_values_pointers[7]=azimuth_values_30_elev;
azimuth_values_pointers[8]=azimuth_values_40_elev;
azimuth_values_pointers[9]=azimuth_values_50_elev;
azimuth_values_pointers[10]=azimuth_values_60_elev;
azimuth_values_pointers[11]=azimuth_values_70_elev;
azimuth_values_pointers[12]=azimuth_values_80_elev;
azimuth_values_pointers[13]=azimuth_values_90_elev;

for (i=0;i<Samples_BufSize;i++)
        {  left_reverb_HRTF[i]=0.0;
         right_reverb_HRTF[i]=0.0;
        }
// calc x,y,z sound source coords in mirrored room
                MirrSource.X = Source.X;
                MirrSource.Y = Source.Y;
                MirrSource.Z = Source.Z;
// calculate distance to listener
        Dist = sqrt((MirrSource.X-Listener.X)*(MirrSource.X-Listener.X)
```

```
+(MirrSource.Y-Listener.Y) *(MirrSource.Y-Listener.Y)+(MirrSource.Z-Listener.Z)*(MirrSource.Z-
Listener.Z));
                    if(Dist<REFDIST)
          Dist=REFDIST;
                    Gain=(REFDIST/Dist)*(REFDIST/Dist);

// compute cartesian coord relative to listener
    MirrSource.X = MirrSource.X - Listener.X;
    MirrSource.Y = MirrSource.Y - Listener.Y;
    MirrSource.Z = MirrSource.Z - Listener.Z;

// compute cartesian coord relative to listener and listener orientation
// theta is angle from (1,0) vector to listener orientation vector
                    float_temp=sqrt((Listener_orientation.X*Listener_orientation.X)
                                        +(Listener_orientation.Y*Listener_orientation.Y));
                    if(float_temp!=0.0)    // orientation nulle ! choisit theta=0
                theta= acos(Listener_orientation.X/float_temp);
    else
                    theta= 0.0;
    if(Listener_orientation.Y < 0)
          theta=-theta; // because theta computation is always between 0 and pi

// compute the rotation
    float_temp=(cos(-theta)*MirrSource.X)+(-sin(-theta)*MirrSource.Y);
    MirrSource.Y=(sin(-theta)*MirrSource.X)+( cos(-theta)*MirrSource.Y);
    MirrSource.X=float_temp;

// compute polar coord relative to listener and listener orientation
                    r=sqrt((MirrSource.X*MirrSource.X)+(MirrSource.Y*MirrSource.Y)
        +(MirrSource.Z*MirrSource.Z));
    if(r!=0.0)              //check distance nulle, si oui phi=0 choisi
                            phi=acos(MirrSource.Z/r);
    else
          phi=0.0;
                    if(MirrSource.X!=0.0)  // check azimuth nul, theta=+-PI/2
              theta=atan(MirrSource.Y/MirrSource.X);
    else
     {   if(MirrSource.Y>=0.0)
                            theta=PI/2;
                    else
                            theta=-PI/2;
     }
    if(MirrSource.X<0)
          theta=theta+PI;//because theta computation is always between 90 and -90
    elevation=((PI/2)-phi)/PI*180.0;
    azimuth=(-theta)/PI*180.0;
    if (azimuth<0)
          azimuth=azimuth+360.0;

//select appropriate HRTFs, read file
// find proper elevation
    float_temp=1000.0;
    for(l=0;l<14;l++)
    { if(fabs(elevation_values[l]-elevation)<fabs(float_temp-elevation))
       {  float_temp=elevation_values[l];
        int_temp=(short int)l;
```

```
        }
    }
    elevation=float_temp;

// find proper azimuth value
    float_temp=0.0;
          for(l=0;l<azimuth_nb_of_values[int_temp];l++)
    {if(fabs(azimuth_values_pointers[int_temp][l]-azimuth)<fabs(float_temp-azimuth))
          float_temp=azimuth_values_pointers[int_temp][l];
    }
    if(fabs(360.0-azimuth)<fabs(float_temp-azimuth))
          float_temp=0.0;
    azimuth=float_temp;

// read hrtf from file
    if(azimuth<=180.0)
      { if(strcmp("diffuse32k",hrtf_sub_path)==0)
                                    sprintf(file_name,"%s\\%s\\elev%d\\H%de%03da.res",
                                    hrtf_path,hrtf_sub_path,(int)elevation,(int)elevation, (int)azimuth);
       else
                                    sprintf(file_name,"%s\\%s\\elev%d\\H%de%03da.dat",
                                    hrtf_path,hrtf_sub_path,(int)elevation,(int)elevation, (int)azimuth);
       if ((hrtf_file_pointer=fopen(file_name,"rb"))==NULL)
                                { sprintf(StrLine,"Unable to open hrtf file : %s",file_name);
                                        MessageBox(NULL,StrLine,"message",
MB_OK|MB_ICONEXCLAMATION);
                                        exit(-1);
                                }
       for(l=0;l<HRTF_LEN;l++)
            {  fread(&int_temp,sizeof(short int),1,hrtf_file_pointer);
               int_temp=((int_temp>>8)&0x00FF)|(int_temp<<8);
                                    file_left_hrtf[l] =((float)int_temp)/32768.0;
                                    fread(&int_temp,sizeof(short int),1,hrtf_file_pointer);
               int_temp=((int_temp>>8)&0x00FF)|(int_temp<<8);
                                    file_right_hrtf[l]=((float)int_temp)/32768.0;
          if(feof(hrtf_file_pointer)!=0)
           {        while(l<HRTF_LEN)
                                                {       file_left_hrtf[l]=0.0;
                                                        file_right_hrtf[l]=0.0;
                                    l++;
                }
           break;
           }
       }
                            fclose(hrtf_file_pointer);
      }
      else
      { azimuth=360.0-azimuth;
                            if(strcmp("diffuse32k",hrtf_sub_path)==0)
                                    sprintf(file_name,"%s\\%s\\elev%d\\H%de%03da.res",
                                    hrtf_path,hrtf_sub_path,(int)elevation,(int)elevation, (int)azimuth);
       else
                                    sprintf(file_name,"%s\\%s\\elev%d\\H%de%03da.dat",
                                    hrtf_path,hrtf_sub_path,(int)elevation,(int)elevation, (int)azimuth);
      if ((hrtf_file_pointer=fopen(file_name,"rb"))==NULL)
                        {  sprintf(StrLine,"Unable to open hrtf file : %s",file_name);
```

```
                                        MessageBox(NULL,StrLine,"message",
MB_OK|MB_ICONEXCLAMATION);
                                        exit(-1);
        }
                        for(l=0;l<HRTF_LEN;l++)
            { fread(&int_temp,sizeof(short int),1,hrtf_file_pointer);
              int_temp=((int_temp>>8)&0x00FF)|(int_temp<<8);
                                file_right_hrtf[l]=((float)int_temp)/32768.0;
                                fread(&int_temp,sizeof(short int),1,hrtf_file_pointer);
              int_temp=((int_temp>>8)&0x00FF)|(int_temp<<8);
                                file_left_hrtf[l] =((float)int_temp)/32768.0;
         if(feof(hrtf_file_pointer)!=0)
                  { while(l<HRTF_LEN)
                                                        { file_right_hrtf[l]=0.0;
                                                                file_left_hrtf[l]=0.0;
                                l++;
                           }
                  break;
             }
        }
        fclose(hrtf_file_pointer);
    }

//contribution to hrtf_eq left and right
          for(l=0;l<HRTF_LEN;l++)
             { left_reverb_HRTF[l]+=(file_left_hrtf[l]*Gain);
       right_reverb_HRTF[l]+=(file_right_hrtf[l]*Gain);
    }

  free(azimuth_values_pointers);
}

/*************************************************************************
OpenWaveFile

This subroutine is responsible for reading in the WAVE file header
from the input file and advancing the file pointer to the data chunk
in the WAVE file. The data format of the input WAVE file is also
checked to ensure it is a type that this program can process (16 bit
mono PCM).
*************************************************************************/
long OpenWaveFile( FILE *InFileBin, PCMWAVEFORMAT *WaveHeader)
{
          char ChunkId[5];
          long ChunkSize;
          InputRiffId( ChunkId, InFileBin);
          if (strcmp(ChunkId,"RIFF")!=0)
          { MessageBox(NULL,"Error: File is not a RIFF file","message",
          MB_OK|MB_ICONEXCLAMATION);
                    exit(-1);
          }
          fread(&ChunkSize,sizeof(ChunkSize),1,InFileBin);
          InputRiffId( ChunkId, InFileBin);
          if (strcmp(ChunkId,"WAVE")!=0)
          { MessageBox(NULL,"Error: File is not a WAVE file","message",
          MB_OK|MB_ICONEXCLAMATION);
```

```
                exit(-1);
        }
        InputRiffId( ChunkId, InFileBin);
        if (strcmp(ChunkId,"fmt ")!=0)
        {
                MessageBox(NULL,"Error: Unable to find fmt chunk","message",
    MB_OK|MB_ICONEXCLAMATION);
                exit(-1);
        }
        fread(&ChunkSize,sizeof(ChunkSize),1,InFileBin);
        fread(WaveHeader,sizeof(*WaveHeader),1,InFileBin);
        InputRiffId( ChunkId, InFileBin);
        if (    ((*WaveHeader).wf.wFormatTag != 1) |
                        ((*WaveHeader).wf.nChannels != 1) |
                        ((*WaveHeader).wBitsPerSample != 16)
                )
                { MessageBox(NULL,"Error: Input file must be a 16 bit mono PCM file",
     "message", MB_OK|MB_ICONEXCLAMATION);
                        exit(-1);
                }
        if (strcmp(ChunkId,"data")!=0)
                { MessageBox(NULL,"Error: Unable to find data chunk",
     "message", MB_OK|MB_ICONEXCLAMATION);
                        exit(-1);
                }
        fread(&ChunkSize,sizeof(ChunkSize),1,InFileBin);
        return(ChunkSize);
}

/*************************************************************************
InputRiffId

This subroutine copies the riff chunk id into a character array.
*************************************************************************/
void InputRiffId( char * ChunkId, FILE * InFileBin)
{  int i;
  for (i=0; i<4; i++)
                fread((ChunkId+i),sizeof(char),1,InFileBin);
        *(ChunkId+4) = \0';}
```

AUDIO.H

```
/*************************************************************************/
/*
/* File Name : audio.h
/* Purpose : Base class for playing audio
/*
/*************************************************************************/
#ifndef AUDIO_H_INCLUDED
#define AUDIO_H_INCLUDED
#include "def.h"
#include <typeinfo>
#include <iostream>
#include <cstddef>
#include <stdlib.h>  // DAN: added because of function exit()
// The following line is necessary if your compiler
//   strictly follows the ANSI C++ Standard (almost none do).
// However, some compilers don't implement this feature at all.
// If your compiler complains about this line,
//   simply comment it out and try again.
//using namespace std;  Adel: removed because of compile error
// typedef short AudioSample; // A single audio sample
// typedef unsigned char AudioByte; // an 8-bit unsigned byte
// The following 7 functions are not used in this application
long ReadIntMsb(istream &in, int bytes);
long BytesToIntMsb(void *buff, int bytes);
long ReadIntLsb(istream &in, int bytes);
long BytesToIntLsb(void *buff, int bytes);
void SkipBytes(istream &in, int bytes);
void WriteIntMsb(ostream &out, long l, int bytes);
void WriteIntLsb(ostream &out, long l, int bytes);
/* AudioAbstract: Base class for playing audio */
class AudioAbstract {
private:
  AudioAbstract *_previous; // object to get data from
  AudioAbstract *_next; // object pulling data from us
public:
  AudioAbstract *Previous(void) { return _previous; }
  void Previous(AudioAbstract *a) { _previous = a; }
  AudioAbstract *Next(void) { return _next; }
  void Next(AudioAbstract *a) {_next = a;}
public:
  AudioAbstract(void) {
    _previous = 0;
    _next = 0;
    _samplingRate = 0;  _samplingRateFrozen = false;
    _channels = 0;      _channelsFrozen = false;
  };
public:
  AudioAbstract(AudioAbstract *audio) {
    _previous = audio;
    _next = 0;
    audio->Next(this);
    _samplingRate = 0;  _samplingRateFrozen = false;
    _channels = 0;      _channelsFrozen = false;
  };
public:
```

```
  virtual ~AudioAbstract(void) { };
public:
  // Returns number of samples actually read, 0 on error.
  // This should always return the full request unless there is
  // an error or end-of-data.
  virtual size_t GetSamples(AudioSample *, size_t) = 0;
public:
  virtual size_t ReadBytes(AudioByte * buff, size_t length) {
    return Previous()->ReadBytes(buff,length);
  };
private:
  long _samplingRate;
  bool _samplingRateFrozen;
public:
  virtual long SamplingRate(void) {
    cerr << "audio.h: SamplingRate(void)\n";
    if (!_samplingRateFrozen)  // Not frozen?
      cerr << "audio.h: NegotiateSamplingRate()\n";
      NegotiateSamplingRate(); // Go figure it out
    return _samplingRate; // Return it
  };
  virtual void SamplingRate(long s) { // Set the sampling rate
    cerr << "audio.h: SamplingRate(long s) with s = " << s << "\n";
    if (_samplingRateFrozen) {
      cerr << "Can't change sampling rate.\n";
      exit(1);
    }
    _samplingRate = s;
  };
public:
  virtual void NegotiateSamplingRate(void);
public:
  virtual void MinMaxSamplingRate(long *min, long *max, long *prefer);
  virtual void SetSamplingRateRecursive(long s);
private:
  long _channels;
  bool _channelsFrozen;
public:
  virtual int Channels(void) {
    if (!_channelsFrozen) NegotiateChannels();
    return _channels;
  };
  virtual void Channels(int ch) {
    if (_channelsFrozen) {
      cerr << "Can't change number of channels.\n";
      exit(1);
    }
    _channels = ch;
  };
  virtual void NegotiateChannels(void);
  virtual void MinMaxChannels(int *min, int *max, int *preferred) ;
  virtual void SetChannelsRecursive(int s);
};
#endif
```

AUDIO.CPP

```
/*
  Copyright 1997 Tim Kientzle.  All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided
that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the
following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the
following disclaimer in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following
acknowledgement:  This product includes software developed by Tim Kientzle and published in "The
Programmer's Guide to Sound."
4. Neither the names of Tim Kientzle nor Addison-Wesley  may be used to endorse or promote products
derived from this software  without specific prior written permission.

THIS SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESS OR IMPLIED
WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
IN NO EVENT SHALL TIM KIENTZLE OR ADDISON-WESLEY BE LIABLE FOR
ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT
OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE
OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*/
/*************************************************************************/
/*
/* FILE NAME : AUDIO.CPP
/*
/* Purpose : Definitions for AudioAbstract class. Complements audio.h
/*
/* Comments added by : Adel Senki (1646856)
/*
/* Date : Dec 15, 1999
/*
/*************************************************************************/
#include "audio.h"

void AudioAbstract::NegotiateSamplingRate(void) {
  if (Next()) // Are we the leftmost?
    Next()->NegotiateSamplingRate(); // No, keep goin
  else { // Yes, we are
    long min = 8000, max = 44100, preferred = 44100;
    MinMaxSamplingRate(&min,&max,&preferred); // Get preferred values
    if (min > max) { // Check for rediculous answers
      cerr << "Couldn't negotiate sampling rate.\n";
      exit(1);
    }
    SetSamplingRateRecursive(preferred); // Set them everywhere
  }
}
```

```
void AudioAbstract::MinMaxSamplingRate(long *min, long *max,
                                       long *preferred) {
  if (Previous()) Previous()->MinMaxSamplingRate(min,max,preferred);
  if (_samplingRate) *preferred = _samplingRate;
  if (*preferred < *min) *preferred = *min;
  if (*preferred > *max) *preferred = *max;
}

void AudioAbstract::SetSamplingRateRecursive(long s) {
  if (Previous()) // Set towards the right first
    Previous()->SetSamplingRateRecursive(s);
  SamplingRate(s); // Set it
  _samplingRateFrozen = true; // Yes, we've negotiated
}
void AudioAbstract::NegotiateChannels(void) {
  if (Next())
    Next()->NegotiateChannels();
  else {
    int min=1, max=2, preferred=1; // Some reasonable default
    MinMaxChannels(&min,&max,&preferred);
    if (min > max) {
      cerr << "Couldn't negotiate sampling rate.\n";
      exit(1);
    }
    SetChannelsRecursive(preferred);
  }
}

void AudioAbstract::MinMaxChannels(int *min, int *max, int *preferred) {
  if (Previous())  Previous()->MinMaxChannels(min,max,preferred);
  if (_channels) *preferred = _channels;
  if (*preferred < *min) *preferred = *min;
  if (*preferred > *max) *preferred = *max;
}

void AudioAbstract::SetChannelsRecursive(int ch) {
  if (Previous()) Previous()->SetChannelsRecursive(ch);
  Channels(ch);
  _channelsFrozen = true;
}

long ReadIntMsb(istream &in, int size) {
  if (size <= 0) return 0;
  long l = ReadIntMsb(in,size-1) << 8;
  l |= static_cast<long>(in.get()) & 255;
  return l;
}

long BytesToIntMsb(void *vBuff, int size) {
  unsigned char *buff = reinterpret_cast<unsigned char *>(vBuff);
  if (size <= 0) return 0;
  long l = BytesToIntMsb(buff,size-1) << 8;
  l |= static_cast<long>(buff[size-1]) & 255;
  return l;
}
```

```
long ReadIntLsb(istream &in, int size) {
  if (size <= 0) return 0;
  long l = static_cast<long>(in.get()) & 255;
  l |= ReadIntLsb(in,size-1)<<8;
  return l;
}

long BytesToIntLsb(void *vBuff, int size) {
  unsigned char *buff = reinterpret_cast<unsigned char *>(vBuff);
  if (size <= 0) return 0;
  long l = static_cast<long>(*buff) & 255;
  l |= BytesToIntLsb(buff+1,size-1)<<8;
  return l;
}

void SkipBytes(istream &in, int size) {
  while (size-- > 0)
    in.get();
}
void WriteIntMsb(ostream &out, long l, int size) {
  if (size <= 0) return;
  WriteIntMsb(out, l>>8, size-1); // Write MS Bytes
  // Adel: changed following line from out.put(l&255)
  out.put(static_cast<char>(l&255)); // Write LS Byte
}

void WriteIntLsb(ostream &out, long l, int size) {
  if (size <= 0) return;
  // Adel: changed following line from out.put(l&255)
  out.put(static_cast<char>(l&255));  // Write LS Byte
  WriteIntLsb(out, l>>8, size-1); // Write rest      }
```

Audiosource.H

```
/*****************************************************************************/
/* File Name : audiosource.h
/* Purpose : Declaration of DynamicAudioSource class for generating audio
/*         samples. This class is a derived class of the AudioAbstract class
/*         from audio.h
/* Written by : Adel Senki (1646856)
/* Date : May 25, 1999
/*****************************************************************************/
#ifndef AUDIOSOURCE_H_INCLUDED
#DEFINE AUDIOSOURCE_H_INCLUDED
#include "audio.h"
#include <iostream.h>
#include <stdlib.h>
#ifndef SAMPLE16_DEFINED
#define SAMPLE16_DEFINED
typedef short Sample16;
#endif
// Derived class of AudioAbstract class
class DynamicAudioSource: public AudioAbstract {
protected:
  long GenWaveOutSamples(Sample16 * pDest, long destSize);
  bool _endOfSource;  // true -> last data read from source
private:
    int                    j,i,int_temp;
  // Total number of audio samples kept for processing
 //volatile unsigned long _samplesKept;
  size_t GetSamples(AudioSample * buffer, size_t numSamples) {
    cerr << "audiosource.h: Error! Not supposed to call GetSamples() here.\n";
    exit(1);
    return (0);
  };
  // ReadBytes: not used (ignore)
  size_t ReadBytes(AudioByte * buffer, size_t numSamples) {
    cerr << "audiosource.h: Error! Not supposed to call ReadBytes() here.\n";
    exit(1);
    return (0);
  };
public:
 DynamicAudioSource(long samplingRate) : AudioAbstract() {
    _endOfSource = false;
 //  _samplesKept = 0;
  };
  ~DynamicAudioSource() { };
  virtual void Play() = 0;
};
#endif
```

Audiosource.cpp

```
/*****************************************************************************/
/*
/* File Name : audiosource.cpp
/*
/* Purpose : Definitions for DynamicAudioSource class. Complements
/*        audiosource.h.
/*        To change the type of audio samples to be generate (i.e.,
/*        sinusoidal samples or white noise samples), simply comment out the
/*        GenWaveOutSamples functions (need two functions in each case; one
/*        for Sample16 and one for Sample8) that you do not want to use and
/*        comment in the GenWaveOutSamples functions to be used. Currently,
/*        the functions for generating sinusoidal samples are in effect
/*        (commented out.)
/*
/* Written by : Adel Senki (1646856)
/*
/* Date : May 25, 1999
/*
/*****************************************************************************/
#include <stdio.h>
#include "audiosource.h"
#include "fourier.h"

unsigned long TotalElapsedSamps=0;                    // elapsed time since beginning
 unsigned long TotalSamplesToRead=0;
 FILE     *InFileBin;
 float            *left_reverb_HRTF;
  float           *im_l_reverb_HRTF;
  float           *out_l_reverb_HRTF;
  float           *imgout_l_reverb_HRTF;
  float           *right_reverb_HRTF;
  float           *im_r_reverb_HRTF;
  float           *out_r_reverb_HRTF;
  float           *imgout_r_reverb_HRTF;
  float           *im_input ;
  float           *out_Sample;
  float           *imgout_Sample;
  long                    Samples_BufSize;
  short            Sample[samsize];                    // variable for reading from file
  float           *Samples;
  float           *out_L_Sample;
  float           *imgout_L_Sample;
  float           *out_R_Sample;
  float           *imgout_R_Sample;
  unsigned                Total_sam;
  float           *Output_left;              // the value of the current output
  float           *Output_right;
  float           *imgout_left;              // the value of the current output
  float                   *imgout_right;

long DynamicAudioSource::GenWaveOutSamples(Sample16 * pDest, long destSize) {

// over lap
    for (long i=0; i<(Samples_BufSize-1); i++)
```

```
    {    out_L_Sample[i]=out_L_Sample[(Total_sam-Samples_BufSize-1)];
         out_R_Sample[i]=out_R_Sample[(Total_sam-Samples_BufSize-1)];
      Total_sam++; }
  // read input
  fread(&(Sample[Samples_BufSize]),sizeof(long),destSize,InFileBin);

  for (long i=0; i<Samples_BufSize; i++)
         Samples[i]=(float)Sample[i] ;

         fft_float(Total_sam,0,Samples,im_input ,out_Sample,imgout_Sample);

  for (long i=0; i<Total_sam; i++)
         {Output_left[i]=(out_Sample[i]*out_l_reverb_HRTF[i])-
(imgout_Sample[i]*imgout_l_reverb_HRTF[i]);
     Output_right[i]=(out_Sample[i]*out_r_reverb_HRTF[i])-
(imgout_Sample[i]*imgout_r_reverb_HRTF[i]);
     imgout_left[i]=(imgout_Sample[i]*imgout_l_reverb_HRTF[i])-
(imgout_Sample[i]*out_l_reverb_HRTF[i]);
     imgout_right[i]=(imgout_Sample[i]*imgout_r_reverb_HRTF[i])-
(imgout_Sample[i]*out_r_reverb_HRTF[i]);
         }

    fft_float(Total_sam,1,Output_left,imgout_left ,out_L_Sample,imgout_L_Sample);
    fft_float(Total_sam,1,Output_right,imgout_right ,out_R_Sample,imgout_R_Sample);
     for (long i=Samples_BufSize; i<destSize; i++) {
                  pDest[2*i] = static_cast <Sample16>(out_L_Sample[2*i] );
          pDest[2*i+1] = static_cast <Sample16>(out_R_Sample[2*i+1]);
    }
      TotalElapsedSamps+=destSize;
      return destSize;
  }
```

WINPLAYR.H

```
/*****************************************************************************/
/* File Name : winplayr.h
/* Purpose : Definitions of class WinPlayer for playing audio. This class is a
/*         derived class from the DynamicAudioSource class, which in turn is
/*         derived from the AudioAbstract class.
/* Comments added by : Adel Senki (#1646856)
/* Date : January 5, 2000
/*****************************************************************************/
//* Player class for playing audio using Win32 MAPIs
#IFNDEF WIN_PLAYER_H_INCLUDED
#define WIN_PLAYER_H_INCLUDED
#include "def.h"
#include "audio.h"
#include "audiosource.h"
#include <windows.h>
#include <mmsystem.h>
extern AudioSample * pBufPlayed;
extern bool DSPReady_PlayedBufferSaved;
extern bool DSPReady;
// Derived class of DynamicAudioSource class
// (and hence of AudioAbstract class also)
class WinPlayer : public DynamicAudioSource {
private:
          unsigned long  _buffer_ID;
  unsigned long  _samplesGenerated;
  HWAVEOUT       _device;  // Windows audio device to open
  volatile bool  _paused; // true -> device is paused
  int            _sampleWidth; // width of data to output
  int SelectDevice(void); // Open a suitable WaveOut device
  /* Callback function called by WaveOut device when it finishes playing a  buffer */
  // Allow the callback to see our members
  friend void CALLBACK WaveOutCallback(HWAVEOUT hwo, UINT uMsg,
          DWORD dwInstance, DWORD dwParam1, DWORD dwParam2);
  // The callback function defined above is just a wrapper that
  // invokes this method
  void NextBuff(WAVEHDR *);
public:
  WinPlayer(long samplingRate,int channels)://, float fSource) :
  DynamicAudioSource(samplingRate) {//, fSource) {
    this->SamplingRate(samplingRate);
 //   this->channels(channels);
                    _buffer_ID=0;
    _samplesGenerated = 0;
    _device = 0;
    _paused = true;
    _sampleWidth = 0;
  };
  ~WinPlayer() {};
  void Play(); // Actually play audio
};
#endif
```

WINPLAYR.CPP

```
/*
  Copyright 1997 Tim Kientzle.  All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided
that the following conditions aremet:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the
following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the
following disclaimer in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following
acknowledgement: This product includes software developed by Tim Kientzle and published in "The
Programmer's Guide to Sound."
4. Neither the names of Tim Kientzle nor Addison-Wesley may be used to endorse or promote products
derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,
INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL TIM
KIENTZLE OR ADDISON-WESLEY BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED
TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*/
/*****************************************************************************/
/*
/* File Name : winplayr.cpp
/*
/* Purpose : Definitions for WinPlayr class. Complements winplayr.h
/*
/* Comments added by : Adel Senki (#1646856)
/*
/* Date : January 5, 1999
/*
/*****************************************************************************/
#include <windows.h>
#include <mmsystem.h>
#include <iostream>
#include "winplayr.h"

void WinPlayer::Play(void) {
  if (SelectDevice()) return; // Open a suitable device
  waveOutPause(_device); // Don't start playing yet
  _paused = true;

//  InitializeQueue(128*1024L); // Allocate 128k queue

  WAVEHDR waveHdr[2];
  for (int i=0; i<2; i++) {
    waveHdr[i].dwBufferLength  // Size in bytes
        = winBufferSize * _sampleWidth/8;
```

```
    waveHdr[i].dwFlags = 0;
    waveHdr[i].dwLoops = 0;
    waveHdr[i].lpData
        = reinterpret_cast<LPSTR>(
          new BYTE[waveHdr[i].dwBufferLength * Channels()]);
    waveOutPrepareHeader(_device,&waveHdr[i],sizeof(waveHdr[i]));
    NextBuff(&waveHdr[i]); // Fill and write buffer to output
  }
  // Wait until finished and both buffers become free
  _paused = false;
  waveOutRestart(_device); // Start playing now

/*WHDR_DONE
Set by the device driver to indicate that it is finished with the buffer
and is returning it to the application.  */

  while(!_endOfSource  // source done??
      || ((waveHdr[0].dwFlags & WHDR_DONE) == 0) // buffers finished?
      || ((waveHdr[1].dwFlags & WHDR_DONE) == 0)) {

    Sleep(50 /* ms */); // Loop about 20 times a second
  }
  MMRESULT err = waveOutClose(_device);
  while (err == WAVERR_STILLPLAYING) { // If it's still playing...
    Sleep(250); // Wait for a bit...
    err = waveOutClose(_device); // try again...      // DAN: added err =
  };

  for(int i1=0; i1<2; i1++) {
    waveOutUnprepareHeader(_device,&waveHdr[i1],sizeof(waveHdr[i1]));
    delete [] waveHdr[i1].lpData;
  }
}
// CallBack
void CALLBACK WaveOutCaliback(HWAVEOUT hwo, UINT uMsg,
  DWORD dwInstance, DWORD dwParam1, DWORD dwParam2) {
  WinPlayer *me = reinterpret_cast<WinPlayer *>(dwInstance);
  switch(uMsg) {
  case WOM_DONE: // Done with this buffer
    {
      WAVEHDR *pWaveHdr = reinterpret_cast<WAVEHDR *>(dwParam1);
      /* dwParam1: address of a WAVEHDR structure identifying the buffer in
        which to put the data */
      me->NextBuff(pWaveHdr);
      break;
    }
  default:
    break;
  }
}
void WinPlayer::NextBuff(WAVEHDR *pWaveHdr) {
  long samplesRead = 0;
   switch(_sampleWidth) {
  case 16:
    samplesRead = GenWaveOutSamples(
        reinterpret_cast<Sample16 *>(pWaveHdr->lpData),
```

```
        winBufferSize);
      break;
    case 8:
//      samplesRead = GenWaveOutSamples(
//          reinterpret_cast<Sample8 *>(pWaveHdr->lpData),
//          winBufferSize);
      break;
    }
    if (samplesRead != 0) {  // I got data, so write it
      // Keep current output buffer if processingDone
      pWaveHdr->dwBufferLength = samplesRead * _sampleWidth / 8;
      waveOutWrite(_device, pWaveHdr, sizeof(*pWaveHdr));
    }
    else if (!_endOfSource) { // Whoops! Source couldn't keep up
      waveOutPause(_device); // pause the output
      _paused = true;
      cerr << "Sound output paused due to lack of data.\n";
      /* Write some zeros to keep this block in Windows' queue.
        The buffer passed back to the application must be written back to the
        system otherwise that buffer is effectively dead (no longer in
        Window's queue)
      */
      memset(pWaveHdr->lpData,0,winBufferSize);
      pWaveHdr->dwBufferLength = 256;
      waveOutWrite(_device,pWaveHdr,sizeof(*pWaveHdr));
    }
    else { // No data, everything's done.

      // Mark buffer as finished but don't write it
      pWaveHdr->dwFlags |= WHDR_DONE;
    }
}
// These are the primary formats supported by Windows
static struct {
  DWORD format; // Constant
  UINT rate;   // break down for this constant
  UINT channels;
  UINT width;
} winFormats[] = {
  {WAVE_FORMAT_1S16, 11025, 2, 16},
  {WAVE_FORMAT_1S08, 11025, 2, 8},
  {WAVE_FORMAT_1M16, 11025, 1, 16},
  {WAVE_FORMAT_1M08, 11025, 1, 8},
  {WAVE_FORMAT_2S16, 22050, 2, 16},
  {WAVE_FORMAT_2S08, 22050, 2, 8},
  {WAVE_FORMAT_2M16, 22050, 1, 16},
  {WAVE_FORMAT_2M08, 22050, 1, 8},
  {WAVE_FORMAT_4S16, 44100, 2, 16},
  {WAVE_FORMAT_4S08, 44100, 2, 8},
  {WAVE_FORMAT_4M16, 44100, 1, 16},
  {WAVE_FORMAT_4M08, 44100, 1, 8},
  {0,0,0,0}
};
//
// Negotiate the sound format and open a suitable output device
int WinPlayer::SelectDevice(void) {
```

```
// Get everyone else's idea of format
int channelsMin = 2, channelsMax = 2, channelsPreferred = 2;
long rateMin = 8000, rateMax = 44100, ratePreferred = 22050;

MinMaxChannels(&channelsMin,&channelsMax,&channelsPreferred);
if (channelsMin > channelsMax) {
  cerr << "Couldn't negotiate channels.\n";
  exit(1);
}

MinMaxSamplingRate(&rateMin,&rateMax,&ratePreferred);
if (rateMin > rateMax) {
  cerr << "Couldn't negotiate rate.\n";
  exit(1);
}

// First, try for an exact match
static const int NO_MATCH=100000;
UINT matchingDevice = NO_MATCH;
WAVEFORMATEX waveFormat;
waveFormat.wFormatTag = WAVE_FORMAT_PCM;
waveFormat.nChannels = channelsPreferred;
waveFormat.nSamplesPerSec = ratePreferred;
waveFormat.wBitsPerSample = 8 * sizeof(Sample16);
waveFormat.nBlockAlign = waveFormat.nChannels
        * waveFormat.wBitsPerSample / 8;
waveFormat.nAvgBytesPerSec = waveFormat.nBlockAlign
        * waveFormat.nSamplesPerSec;
waveFormat.cbSize = 0;
MMRESULT err = waveOutOpen(0,WAVE_MAPPER,&waveFormat,
                0,0,WAVE_FORMAT_QUERY);
if (err == 0) {
  matchingDevice = WAVE_MAPPER;
  channelsMax = channelsMin = channelsPreferred;
  rateMax = rateMin = ratePreferred;
  _sampleWidth = 16;
} else {
  cerr << "WinPlay: Custom format failed, ";
  cerr << "trying standard formats.\n";
}

// Get count of available devices
UINT numDevs = waveOutGetNumDevs();
if (numDevs == 0) {
  cerr << "No sound output devices found!?\n";
  exit(1);
}
// Check each available device
for (UINT i=0; (i<numDevs) && (matchingDevice == NO_MATCH); i++) {
  // What formats does this device support?
  WAVEOUTCAPS waveOutCaps;
  MMRESULT err =
        waveOutGetDevCaps(i,&waveOutCaps,sizeof(waveOutCaps));
  if (err != MMSYSERR_NOERROR) {
    cerr << "Couldn't get capabilities of device " << i << "\n";
    continue;
```

```
    }
    // Check each standard format
    for(UINT j=0; winFormats[j].format != 0; j++) {
      if ((winFormats[j].format & waveOutCaps.dwFormats) // supported?
        &&(rateMin <= winFormats[j].rate) // Rate ok?
        &&(rateMax >= winFormats[j].rate)
        &&(channelsMin <= winFormats[j].channels) // channels ok?
        &&(channelsMax >= winFormats[j].channels)) {

        // Set up my parameters
        matchingDevice = i;
        rateMin = rateMax = ratePreferred = winFormats[j].rate;
        channelsPreferred = winFormats[j].channels;
        channelsMin = channelsMax = channelsPreferred;
        _sampleWidth = winFormats[j].width;

        // Set up WAVEFORMATEX structure accordingly
        waveFormat.wFormatTag = WAVE_FORMAT_PCM;
        waveFormat.nChannels = winFormats[j].channels;
        waveFormat.nSamplesPerSec = winFormats[j].rate;
        waveFormat.wBitsPerSample = winFormats[j].width;
        waveFormat.nBlockAlign = waveFormat.wBitsPerSample / 8
                    * waveFormat.nChannels;
        waveFormat.nAvgBytesPerSec = waveFormat.nBlockAlign
                     * waveFormat.nSamplesPerSec;
        waveFormat.cbSize = 0;
      }
    }
  }
  if (matchingDevice == NO_MATCH) {
    cerr << "Can't handle this sound format.\n";
    cerr << "Rate: " << rateMin << "-" << rateMax << "\n";
    cerr << "Channels: " << channelsMin << "-" << channelsMax << "\n";
    return 1;
  }

  // If we found a match, set everything
  SetChannelsRecursive(channelsPreferred);
  SetSamplingRateRecursive(ratePreferred);

  // Open the matching device
  MMRESULT err2 = waveOutOpen(&_device,matchingDevice,
    &waveFormat, reinterpret_cast<DWORD>(WaveOutCallback),
    reinterpret_cast<DWORD>(this),CALLBACK_FUNCTION);
  if (err2) {
    cerr << "Couldn't open WAVE output device.\n";
    exit(1);
  }

  return 0;
}
```

fourierf.c

```
/*=====================================================================
   fourierf.c - Don Cross <dcross@intersrv.com>
   http://www.intersrv.com/~dcross/fft.html
   Contains definitions for doing Fourier transforms
   and inverse Fourier transforms.
   This module performs operations on arrays of 'float'.
   Revision history:
1998 September 19 [Don Cross]
   Updated coding standards.
   Improved efficiency of trig calculations.
=====================================================================*/
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <fourier.h>
#include <ddcmath.h>
#define CHECKPOINTER(p)  CheckPointer(p,#p)
static void CheckPointer ( void *p, char *name )
{
   if ( p == NULL )
   {
      fprintf ( stderr, "Error in fft_float():  %s == NULL\n", name );
      exit(1);
   }
}
void fft_float (
   unsigned  NumSamples,
   int       InverseTransform,
   float     *RealIn,
   float     *ImagIn,
   float     *RealOut,
   float     *ImagOut )
{
   unsigned NumBits;    /* Number of bits needed to store indices */
   unsigned i, j, k, n;
   unsigned BlockSize, BlockEnd;
   double angle_numerator = 2.0 * DDC_PI;
   double tr, ti;    /* temp real, temp imaginary */
   if ( !IsPowerOfTwo(NumSamples) )
   {
      fprintf (
         stderr,
         "Error in fft():  NumSamples=%u is not power of two\n",
         NumSamples );
      exit(1);
   }
   if ( InverseTransform )
      angle_numerator = -angle_numerator;
   CHECKPOINTER ( RealIn );
   CHECKPOINTER ( RealOut );
   CHECKPOINTER ( ImagOut );
   NumBits = NumberOfBitsNeeded ( NumSamples );
 //   **  Do simultaneous data copy and bit-reversal ordering into outputs...
for ( i=0; i < NumSamples; i++ )
```

```
    {
        j = ReverseBits ( i, NumBits );
        RealOut[j] = RealIn[i];
        ImagOut[j] = (ImagIn == NULL) ? 0.0 : ImagIn[i];
    }
//**  Do the FFT itself...
    BlockEnd = 1;
    for ( BlockSize = 2; BlockSize <= NumSamples; BlockSize <<= 1 )
    {
        double delta_angle = angle_numerator / (double)BlockSize;
        double sm2 = sin ( -2 * delta_angle );
        double sm1 = sin ( -delta_angle );
        double cm2 = cos ( -2 * delta_angle );
        double cm1 = cos ( -delta_angle );
        double w = 2 * cm1;
        double ar[3], ai[3];
        double temp;
        for ( i=0; i < NumSamples; i += BlockSize )
        {
            ar[2] = cm2;
            ar[1] = cm1;
            ai[2] = sm2;
            ai[1] = sm1;
            for ( j=i, n=0; n < BlockEnd; j++, n++ )
            {
                ar[0] = w*ar[1] - ar[2];
                ar[2] = ar[1];
                ar[1] = ar[0];
                ai[0] = w*ai[1] - ai[2];
                ai[2] = ai[1];
                ai[1] = ai[0];
                k = j + BlockEnd;
                tr = ar[0]*RealOut[k] - ai[0]*ImagOut[k];
                ti = ar[0]*ImagOut[k] + ai[0]*RealOut[k];
                RealOut[k] = RealOut[j] - tr;
                ImagOut[k] = ImagOut[j] - ti;
                RealOut[j] += tr;
                ImagOut[j] += ti;
            }
        }
        BlockEnd = BlockSize;
    }
//  **  Need to normalize if inverse transform...
    if ( InverseTransform )
    {
        double denom = (double)NumSamples;
        for ( i=0; i < NumSamples; i++ )
        {
            RealOut[i] /= denom;
            ImagOut[i] /= denom;
        }
    }
}       /*--- end of file fourierf.c ---*/
```

fftmisc.c

```
/*=========================================================================
   fftmisc.c  - Helper routines for Fast Fourier Transform implementation.
   Contains common code for fft_float() and fft_double().
   See also:   fourierf.c,  fourierd.c, and    ..\include\fourier.h
     =========================================================================*/
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <fourier.h>
#define TRUE  1
#define FALSE 0
#define BITS_PER_WORD  (sizeof(unsigned) * 8)
int IsPowerOfTwo ( unsigned x )
{
   if ( x < 2 )
      return FALSE;
   if ( x & (x-1) )       // Thanks to 'byang' for this cute trick!
      return FALSE;
   return TRUE;
}
unsigned NumberOfBitsNeeded ( unsigned PowerOfTwo )
{   unsigned i;
   if ( PowerOfTwo < 2 )
   {     fprintf (
         stderr,
         ">>> Error in fftmisc.c: argument %d to NumberOfBitsNeeded is too small.\n",
         PowerOfTwo );
      exit(1);
   }
   for ( i=0; ; i++ )
   {
      if ( PowerOfTwo & (1 << i) )
         return i;
   }
}
unsigned ReverseBits ( unsigned index, unsigned NumBits )
  unsigned i, rev;
   for ( i=rev=0; i < NumBits; i++ )
   {
      rev = (rev << 1) | (index & 1);
      index >>= 1;
   }
   return rev;
}
double Index_to_frequency ( unsigned NumSamples, unsigned Index )
{
   if ( Index >= NumSamples )
      return 0.0;
   else if ( Index <= NumSamples/2 )
      return (double)Index / (double)NumSamples;
   return -(double)(NumSamples-Index) / (double)NumSamples;
}                    /*--- end of file fftmisc.c---*/
```

ddc.h

```
/*=============================================================================
        ddc.h - Don cross, October 1992.
        Generic ddclib stuff.
=============================================================================*/
#ifndef __DDC_DDC_H
#define __DDC_DDC_H
// If you add something to DDCRET, please add the appropriate string
// to the function DDCRET_String() in the file 'source\ddcret.cpp'.
enum DDCRET
{
  DDC_SUCCESS,          // The operation succeded
  DDC_FAILURE,          // The operation failed for unspecified reasons
  DDC_OUT_OF_MEMORY,    // Operation failed due to running out of memory
  DDC_FILE_ERROR,       // Operation encountered file I/O error
  DDC_INVALID_CALL,     // Operation was called with invalid parameters
  DDC_USER_ABORT,       // Operation was aborted by the user
  DDC_INVALID_FILE      // File format does not match
};
const char *DDCRET_String ( DDCRET );  // See source\ddcret.cpp
#define TRUE    1
#define FALSE   0
typedef int dBOOLEAN;
typedef unsigned char BYTE;
typedef unsigned char       UINT8;
typedef signed   char       INT8;
typedef unsigned short int  UINT16;
typedef signed   short int  INT16;
typedef unsigned long  int  UINT32;
typedef signed   long  int  INT32;

#ifdef __BORLANDC__
        #if sizeof(UINT16) != 2
          #error Need to fix UINT16 and INT16
        #endif
        #if sizeof(UINT32) != 4
          #error Need to fix UINT32 and INT32
        #endif
#endif
#endif /* __DDC_DDC_H */
/*--- end of file ddc.h ---*/
```

Appendix C C++ source code for a simple application (to be compiled as an executable) calling the 3-D sound DLL application

```
#include <windows.h>
#include <stdio.h>
struct point
{ float X;
        float Y;
        float Z; };
// structure for a point paired with a time duration
struct position
{ struct point Coord;
        float Time; };
int WINAPI WinMain(HINSTANCE hCurInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int
nCmdShow)

{       char function_output;
        struct position   CurrentSource;  // current source in positions list
//      struct point              Room;                    // coords for the room size
        struct point              Listener;          // coords of the listener
 struct point             Listener_orientation;          // orientation of the listener in a 2-D plane
// int    ClosedPath=1;
char   InFileName[128]="noise44p1k20sec.wav";
char      hrtf_path[256]="c:\\user\\mit_hrtf";
char      hrtf_sub_path[64]="compact";
char (  *playsound_ptr)(char *,char *,char *,struct point,struct point,struct position);
HINSTANCE hLib;
hLib = LoadLibrary("fft_dll.dll");
if(hLib==NULL)
{       MessageBox(NULL, "LoadLibrary not successfull !","ERROR",
MB_OK|MB_ICONEXCLAMATION);
        return(-1);
        }
  playsound_ptr = (char  ( *)(char *,char *,char *,struct point,struct point,struct position))
  GetProcAddress(hLib, "_playsound");
  if(playsound_ptr==NULL)
          { MessageBox(NULL, "GetProcAddress not successfull !!!!!",
                  "ERROR", MB_OK|MB_ICONEXCLAMATION);
    return(-1);
   }
  //Room.X=4.0;Room.Y=4.0;Room.Z=4.0;
  Listener.X=2.0;Listener.Y=2.0;Listener.Z=2.0;
  Listener_orientation.X=1.0;Listener_orientation.Y=0.0;
  CurrentSource.Coord.X=0.0; CurrentSource.Coord.Y=1.0;
  CurrentSource.Coord.Z=4.0; CurrentSource.Time=40.0;
  function_output=playsound_ptr(hrtf_path,hrtf_sub_path,InFileName,
  Listener_orientation,Listener,CurrentSource);
        FreeLibrary(hLib);

        return(0);
}
```