HELSINKI UNIVERSITY OF TECHNOLOGY

Department of Electrical and Communications Engineering

Matti Paavola

Advanced audio interfaces for mobile Java

This Master's Thesis has been submitted for official examination for the degree of Master of Science in Espoo on June 2, 2003.

Supervisor of the Thesis: Professor Matti Karjalainen

Instructor of the Thesis: Antti Rantalahti, M.Sc. (Tech.)

TEKNILLINEN KORKEAKOULU Diplomityön tiivistelmä

Tekijä:
Matti Paavola

Työn nimi:
Edistyneet äänirajapinnat kannettavaan Javaan

Päivämäärä: 2. kesäkuuta 2003 Sivumäärä: 64

Osasto:
Sähkö- ja tietoliikennetekniikan osasto

Professuuri:
S-89 Akustiikka ja äänenkäsittelytekniikka

Työn valvoja:
professori Matti Karjalainen

Työn ohjaaja:
dipl.ins. Antti Rantalahti

Tiivistelmäteksti:

Tämä työ käsittelee kannettavan laitteen äänisignaalinkäsittelyn ohjaamista ohjelmallisesti.

Työssä käydään läpi työn kannalta oleelliset psykoakustiikan ilmiöt ja niiden sovellus: virtuaaliakustiikka. Toisaalta työssä tutustutaan viiteen eri äänisignaalinkäsittelyn ohjausrajapintaan, jotka ovat yleisesti käytössä henkilökohtaisissa tietokoneissa. Lisäksi yksi kannettaviin laitteisiin suunniteltu Java-kielinen rajapinta käydään läpi. Tämä ohjelmointirajapinta muodostaa alustan työssä itse kehitettävälle rajapinnalle.

Työn pääasiallisena tuloksena syntyy uusi äänisignaalinkäsittelyn ohjausrajapinta. Tämä rajapinta on suunniteltu erityisesti kannettaville laitteille. Uusi rajapinta tukee muun muassa 3D-ääntä, keinotekoista kaiuntaa, taajuuskorjausta ja erilaisia tehostesuotimia. Esitelty rajapinta mahdollistaa esimerkiksi virtuaaliakustiikan luomisen käyttäjälle kuultavaksi matkapuhelimeen liitettyjen kuulokkeiden välityksellä. Rajapinnasta tarjotaan Windows-ympäristöön referenssitoteutus ja sen päälle rakennettu esittelysovellus.

Työn tuloksena syntynyttä rajapintaa verrataan työssä läpikäytyihin yleisiin ohjelmointirajapintoihin ja todetaan, että syntynyt rajapinta on kykeneväinen moniin samoihin asioihin kuin suuremmatkin rajapinnat ollen silti kevyempi ja siten soveltuvampi kannettaviin laitteisiin.

Avainsanat:
3D-ääni, J2ME, Java, Mobile Media API, virtuaaliakustiikka

Author:
Matti Paavola

Name of the Thesis:
Advanced audio interfaces for mobile Java

Date: June 2, 2003 Number of Pages: 64

Department:
Department of Electrical and Communications Engineering

Professorship:
S-89 Acoustics and Audio Signal Processing

Supervisor:
Professor Matti Karjalainen

Instructor:
Antti Rantalahti, M.Sc. (Tech.)

Abstract:

The problem addressed in this thesis was how to access and control audio processing features of a modern portable communications device.

As a background, the essential psychoacoustical phenomena and their application in virtual acoustics were reviewed. Five traditional ways to access audio processing features from a computer program were studied and considered. These were different programming interfaces common in desktop computing nowadays. Moreover, one Java interface specifically designed mainly for media playback and recording in mobile devices was also studied. This interface formed the basis for the work done in this thesis.

As the main result, this thesis presents a new interface to control audio processing in a mobile information device. The new interface has ready-made support for various audio processing capabilities, such as 3D audio, artificial reverberation, equalization, and audio processing effects. The interface enables the Java application in the mobile phone, for instance, to create an artificial three-dimensional acoustical world via attached headphones. The novel interface is designed to be both lightweight and easy to use. A reference implementation of the new interface is provided and a demonstration application is build on top of it.

The novel interface and the desktop computing interfaces described in this thesis were compared. The presented new interface designed for the mobile world was found to be capable of doing many of the same things as bigger programming interfaces of the personal computer world.

Keywords:
3D audio, J2ME, Java, Mobile Media API, virtual acoustics

Preface

This work has been done at Nokia Research Center's Speech and Audio Systems Laboratory in Helsinki during years 2002 and 2003.

I would like to thank both my instructor Mr. Antti Rantalahti and my superior at work Dr. Jyri Huopaniemi for valuable comments and ideas during my thesis work. I thank my supervisor at the university Professor Matti Karjalainen for not only supervising this thesis, but also for being a skillful and very inspiring lecturer during my acoustics studies. I also thank my colleagues for giving such a pleasant environment to work in.

Most of all, I would like to thank my mother, relatives, and friends for understanding and supporting me while being almost constantly busy during my studies.

Helsinki, May 29, 2003

Matti Paavola

Table of contents

Abbreviations

AAI	Advanced Audio API
API	Application Programming Interface
AWT	Abstract Windowing Toolkit
BEM	Boundary Element Method
BIFS	Binary Format for Scenes
CDC	Connected Device Configuration
CLDC	Connected Limited Device Configuration
COM	Component Object Model
DSP	Digital Signal Processing
EAX	Environmental Audio eXtensions
ECMA	ECMA International - European association for standardizing information and communication systems. (Formerly known as European Computer Manufacturers Association.)
EQ	Equalizer
ERB	Equivalent Rectangular Bandwidth
FDN	Feedback Delay Networks
FEM	Finite Element Method
FIR	Finite Impulse Response
HAL	Hardware Abstraction Layer
HEL	Hardware Emulation Layer
HRTF	Head-Related Transfer Function
HTML	HyperText Mark-up Language
IEC	International Electrotechnical Commission
IIR	Infinite Impulse Response
ILD	Inter-aural Level difference
ISO	International Organization for Standardization
ITD	Inter-aural Time Difference

J2ME	Java 2 Micro Edition
J2SE	Java 2 Standard Edition
JCP	Java Community Process
JNI	Java Native Interface
JSR	Java Specification Request
JVM	Java Virtual Machine
KVM	K Virtual Machine
LFO	Low Frequency Oscillator
MIDI	Musical Instrument Digital Interface
MIDP	Mobile Information Device Profile
MMAPI	Mobile Media Application Programming Interface
MPEG	Moving Picture Experts Group
PDA	Personal Digital Assistant
SA	Structured Audio
SAOL	Structured Audio Orchestra Language
SASBF	Structured Audio Sample Bank Format
SASL	Structured Audio Score Language
SMIL	Synchronized Multimedia Integration Language
SNHC	Synthetic-Natural Hybrid Coding
T60	Time that is taken for the sound pressure level to attenuate 60 dB. A measure of reverberation time.
UI	User Interface
UML	Universal Modeling Language
URL	Universal Resource Locator
VRML	Virtual Reality Modeling Language
X3D	eXtensible 3D

1. Introduction

The hardware of mobile devices evolves all the time, in the means of processing power, memory amount, and different novel media capabilities, such as playback, processing, and capture for both audio and video. These improvements enable completely new types of applications for mobile devices.

On the other hand, most of the new mobile phones sold today have support for Java applications. The user can download Java applications to his or her phone from the Internet and therefore, customize the services that are available in the mobile terminal. Java Mobile Media Application Programming Interface [MMAPI 2002] enables usage of new audio and video features of a mobile phone from a Java application. MMAPI mainly concentrates on the playback and the capture of media content, but does not have a strong support for media processing.

This thesis presents one suggestion how to improve audio processing capabilities of MMAPI 1.0. A new interface, namely Advanced Audio Application Programming Interface (AAI), is presented. It complements MMAPI 1.0 and has ready-made support for various audio processing capabilities, such as 3D audio processing, artificial reverberation, equalizer, and audio processing effects. This new interface enables, for instance, for the Java application in the mobile phone a possibility to create an artificial three-dimensional acoustical world to the user via attached headphones.

The thesis is divided so that chapters from two to five provide the necessary background information and chapters six and seven describe the own work of the author. At first, chapters two and three introduce the reader to the most important related psychoacoustical phenomena and their application in virtual acoustics. Then, chapter four introduces five essential 3D audio APIs on the personal computing market today. In chapter five, the modules of Java that are nowadays common in the mobile phones are gone through. The novel API of this thesis is handled in chapter six; it is first described, then compared to APIs presented in the previous chapter, and then a reference implementation of it is presented. Chapter seven presents one example application that is built on top of the AAI and chapter eight concludes the thesis.

2. Psychoacoustics

To be able to create artificial acoustical environments that sound like real acoustical environments, one has to understand how human hearing works and what are the essential properties of the sounds what comes to perception. One can then concentrate on modeling just the essential parts of the acoustical world and not to waste resources on unessential acoustic phenomena. The field of science that studies how different physical sound stimuli are mapped to different sensations is called psychoacoustics.

This chapter presents the most important psychoacoustical phenomena. First, the human ear is studied in the physiological point of view and then different properties of hearing are gone through. Books [Blauert 1997], [Moore 1997], and [Karjalainen 1999] act as references.

2.1 Physiology of the ear

2.1.1 The outer and the middle ear

The human ear can be divided into three parts: the outer, middle, and inner ears, as shown in Figure 1. The outer ear is a passive part of the ear and consists of the pinna and the auditory canal. Pinna, together with the head and upper body, colors sounds based on the direction from which they arrive. This is one of the fundamental physical phenomena that directional hearing uses.

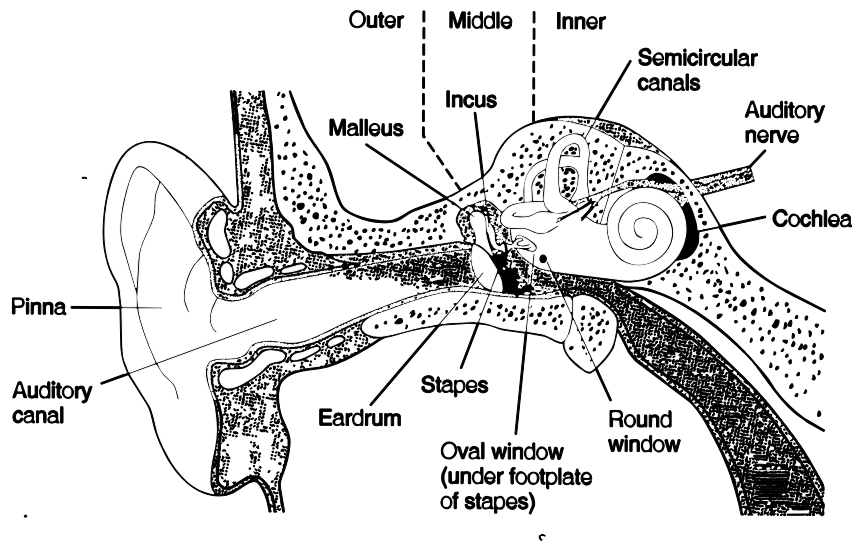

Figure 1. The human ear with its three subdivisions, namely outer, middle, and inner ear, visible. [Goldstein 1999]

The auditory canal leads the sound from the pinna to the eardrum, which separates the middle ear from the outer ear. The auditory canal is a 25 mm long and 7,5 mm wide tube on average. Based on physical dimensions of the auditory canal, it has a resonance that boosts the frequencies around 4 kHz. Because the canal is so narrow, sound in audible frequencies propagates there mainly in one dimension only, as planar waves. Therefore, sounds from different directions are not treated differently anymore after the pinna and thus the auditory canal does not have remarkable influence on spatial hearing.

The middle ear is an air cavity, insulated by the eardrum, containing small bones called the ossicles. The cavity is connected to the back of the throat by the Eustachian tube that balances the static air pressure with the outer ear when, for instance, flying in an airplane. Otherwise, the static air pressure difference would tighten the eardrum making the sensitivity of the hearing poorer.

The role of the ossicles, namely malleus, incus, and stapes, is to propagate the sound from the eardrum to the inner ear. The inner ear is filled with liquids and thus has remarkably different acoustical impedance than air in the auditory canal. Matching these acoustical impedances is an important task of the middle ear. Without it, most of the sound energy would reflect back from the oval window that is the entrance of the inner ear. This mechanical impedance matching is caused both by the large difference in surface areas of the eardrum and the oval window, and by the leverage effect of the ossicles.

The middle ear is not completely passive. It has a small muscle, namely stapedius, attached to stirrup, that contracts when the ear is exposed to loud sound. When the stapedius muscle contracts and pulls the stirrup, the transmission of the sound to the inner ear reduces. The contraction happens after an exposure to sound over about 80 dB in level. Reaction time is some tens or hundreds of milliseconds. The reason for this so called acoustical reflex is probably to try to protect the inner ear from harmful loud sounds. Unfortunately, the acoustical reflex cannot protect the ear well against the harmful sounds of the modern society, because of two reasons. Firstly, the transmission reduction takes place in the lower frequencies only, and secondly, the time lag between the beginnings of the loud sound and the reflex is too long to protect against impulse sounds, such as gunshots.

2.1.2 The inner ear

The ear acts as a sense receptor and converts the acoustical energy it senses to electrical nerve impulses to be transferred via auditory nerve to the brain for further processing. This conversion takes place in the inner ear. Besides of the hearing related functionality, also the vestibular organ is located in the inner ear, but it is out of this study's scope.

The cochlea (Figure 2) is the part of the inner ear where the sensory cells lie. It is a tube around 35 mm in length, and is bent into a spiral form. The cochlea has rigid bony walls and is filled with fluids. The cochlea is split in longitudinal direction with two membranes, namely the basilar and the Reissner's membranes. They divide the cochlea into two larger chambers, namely scala vestibuli and scala tympani, and into one smaller chamber located between them, scala media. The cochlea enlarges in diameter towards the outer end where it is connected with the middle ear. On that end, there are two openings sealed with membranes, the oval window and the round window. Scala vestibuli's fluid is connected via the oval window to the stapes, the small bone that transmits the sound vibration to the inner ear from the middle ear. Scala tympani ends to the round window that leads to the middle ear cavity without similar bone connection. In the inner end of the cochlea there is

an opening called helicotrema in the basilar membrane that connects the fluids of the large chambers, scala vestibuli and scala tympani.

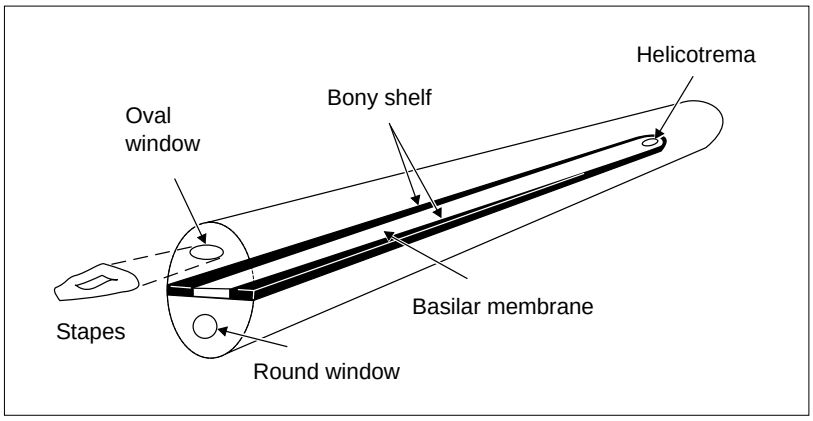

Figure 2. A structural picture of an unfolded cochlea [Karjalainen 1999]. Printed with a permission of the original author.

On the basilar membrane lies the organ of Corti where the receptors, the hair cells, are located. When the sound energy makes the fluids in the cochlea vibrate, also the basilar membrane vibrates and makes the hair cells bend. This causes them to generate electrical impulses to the auditory nerve and to transmit the information towards the brain.

The physical properties of the basilar membrane change as the function of place: when going towards the inner narrower end of the cochlea, the basilar membrane gets wider, heavier, and more flexible. This makes the frequency-dependent sensitivity of the membrane change over the distance from the windows. In other words, the amplitude maximum of the basilar membrane's vibration is located in a frequency dependent place. Hair cells closer to the windows get more excitation for high-frequency sounds and cells closer to the inner end for low-frequency sounds. This is one reason why humans can sense the pitch of sound.

Not all the hair cells are receptors; the outer hair cells have also an effector nature and can cause the basilar membrane to vibrate. It is assumed that these active cells make the resonance peak of the membrane sharper and thus improve ear's frequency selectivity as well as sensitivity.

Another thing that probably improves the sensitivity of pitch analysis is that the nerve impulses transmitted from the hair cells are statistically synchronized to the waveform of the affecting sound. The cells mainly emit pulses during the positive half-wave of the sound (increased pressure) and not during the negative half-wave.

2.2 Critical bands and masking

Critical bands play a central role in hearing. The human auditory system can be divided to critical bands in the frequency domain. When two pure tones are so close in frequency that there is considerable overlap in their amplitude envelopes on the basilar membrane, they are said to lie within the same critical band. About 24 critical bands span the audible frequency range. Their bandwidth varies as a function of their center frequency so that at lower frequencies the bandwidth is around constant 100 Hz, but at around 500 Hz the bandwidth starts to rise logarithmically, being several kilohertz wide in the highest audible frequencies. Critical bands have to be understood so that their location in the frequency

scale is not fixed, but rather so that hearing treats different sounds laying within one critical bandwidth together in various ways.

One scale of pitch, namely the *Bark*-scale, is formed by attaching critical bandwidths next to each other on the frequency scale. One Bark then corresponds one critical bandwidth. The other scale addressing the same needs as the Bark-scale is the ERB-rate scale. It mainly differs from the Bark-scale by the method how the analysis bandwidth of the hearing is measured. The actual subjective pitch scale, the mel-scale, is presented in a separate section later.

Masking is one phenomenon where critical bands play an important role. When the ear is exposed to two or more different tones, one of these tones may mask the others. This happens in the frequency domain between tones inside the same critical band and partially for neighboring bands, and in the time domain between tones that are temporally near each other, so that even a loud sound after a weaker one can mask the other. This premasking happens only in a period starting 5-10 ms before the masking tone starts. After a masking tone ends, postmasking masks tones until 150-200 ms has passed.

2.3 Loudness

In the physical world, the sound pressure means how much air pressure maximally alters from static air pressure when sound wave propagates via one point. The sound pressure is measured in Pascal scale (Pa = N/m^2) and the range of human hearing is of magnitude from 10^{-5} Pa to 10^2 Pa. The range being so large, it is convenient to use a different scale: sound pressure level L_p in decibels (dB). $L_p = 20\log_{10}(p/p_0)$, where p_0 is the reference value 20µPa. This decibel scale also happens to be closer to how human perceives the loudness of sound than Pascal scale.

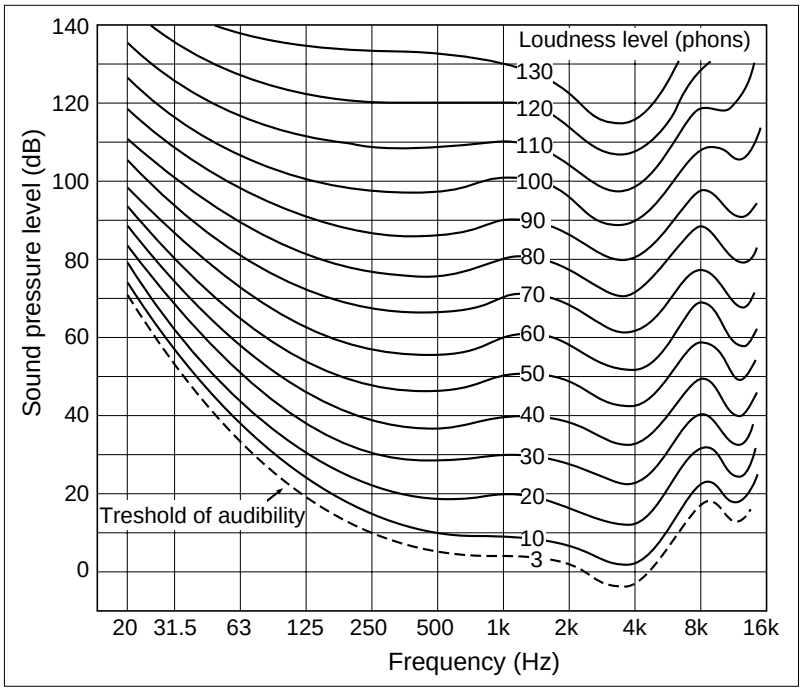

Figure 3. Equal-loudness curves after ISO 226:1987 [Karjalainen 1999]. Printed with a permission of the original author.

Loudness is strongly related to sound pressure level, but also the other magnitudes of sound, such as frequency, spectral distribution, and duration, affect the perceived loudness. Frequency dependency of loudness can be seen on Figure 3, where equal-loudness curves are presented. On the curve, the perceived loudness stays the same. Loudness level is defined in *phons*, so that at 1kHz frequency, the level in phons is numerically equal to sound pressure level in decibels. Loudness level of one or more sounds in phons can furthermore be converted to loudness in *sones* to actually present the overall perception.

Phons and sones are quite rarely used and sound pressure level in decibels is much more common. The sound pressure level is usually given using some frequency dependent weighting to represent human's sensation a little better than a non-weighted value. A-weighting curve is the most commonly used weighting. It approximates roughly equal-loudness curve of 30 phons.

2.4 Pitch

Perceived pitch is strongly related to physical frequency, but like in the case of loudness, it is a combination of also other properties of the sound. The unit used for subjective pitch is the *mel*. Doubling the number of mels, pitch is perceived twice as high. From 0 to 2400 mels span the frequency range from 0 to 16 kHz, and 100 mels corresponds roughly one critical bandwidth (one Bark). Mel scale is one example of the important role the critical bands play in hearing.

Like in the case of phons and sones, mels are not widely used and logarithmic frequency scale is used instead in technical applications. Logarithmic frequency scale is anyway closer to pitch perception than the normal linear frequency scale.

2.5 Timbre and coloration

A piano sound and a violin sound can have equal loudness and pitch, but they still sound different. This is because their frequency components are differently spread on the spectrum and time: they have different timbre.

Humans can hear quite easily changes in timbre. We say then that the sound is colored. Colorations on sound are actually changes in the frequency response of the system. A just noticeable difference in the spectrum of a signal is approximately 1 dB in each critical band (1 Bark), if the reference is on the short time memory of the listener. If time has passed (hours or days) since reference, the errors in the spectrum can be up to ±5-10 dB until they are perceived clearly.

2.6 Localization

2.6.1 Localization cues

Listener's two main cues for localizing a sound source are the interaural time difference (ITD) and the interaural level difference (ILD). ITD is caused by the wave propagation time difference, and ILD by the shadowing effect of the head. ITD is significant primarily below 1.5 kHz and ILD primarily above 1.5 kHz. These cues tell in which cone of

confusion the source lies. The cones of confusion are roughly speaking cones that have a line between ears as their symmetry axis. In other words, on the cone the difference between distances from the sound source to the left ear and to the right ear is the same.

Human can differentiate if the sound source is on the back or on the front, so still one cue is needed. The location on the cone is revealed by spectral cues of the sound that are caused by the physiology of upper part of the human body, mainly by pinnae, but also, for instance, by shoulders. Reflections from different body parts colorize the sound differently depending on the arrival direction.

All these location cues can be presented as head-related transfer functions (HRTF) that present how the sound changes when arriving from a direction to the ear canal. HRTFs are of individual nature, because all humans have different anatomy of head and torso.

2.6.2 Precedence effect

What if the same sound comes from multiple directions? This is a quite common situation that happens, for example, when the sound comes directly from the source and also via a reflection. If the sounds come really close together in the time domain, so that the second sound follows within 1-1.5 ms, they are perceived as a single source, and the location is determined somewhere between two real sources, based on time and level differences between these real sources. If it takes a little bit longer time for the second source's sound to arrive to ears, the location is determined almost entirely based on the first source. So, the ear is practically deaf to the second source when it comes to location, and "considers" this second sound to be an early reflection. This is called the precedence effect, and it takes place until 30-40 ms. After that, the second sound is perceived as an echo or reverberation. Although the secondary sound does not affect the location if the delay is between circa 1.5 ms and 30 ms, it still has an effect on coloration.

3. Virtual Acoustics

"Virtual acoustics is a general term for the modeling of acoustical phenomena and systems with the aid of a computer." [Huopaniemi 1999] This chapter introduces first virtual acoustics modeling, and then reproduction techniques. Modeling can be divided into three tasks: source, transmission medium, and receiver modeling. Each of them is presented in separate subchapters and then ways to reproduce the modeled audio scene are introduced. The last subchapter handles implementation issues.

[Huopaniemi 1999] and [Savioja 1999] act as references in this chapter. Parts of the text have been previously published in [Kujala, Paavola 2002].

3.1 Source modeling

The most straightforward way to model an audio source in a virtual acoustical scene is to use pre-recorded monophonic digital audio and then treat it as an omni-directional point source. The recorded audio should be anechoic, that is, not containing any reflections or reverberation; the room modeling part is responsible for creating these effects in virtual acoustics. High quality anechoic recordings can be done in an anechoic chamber. Instead of using natural pre-recorded digital audio, also synthetic audio can be used, thus saving bandwidth. Furthermore, audio synthesis techniques usually produce anechoic data.

The omni-directional point source model is usually used, but there are also more accurate models available. Natural sound sources are not omni-directional in general. They have a frequency-dependent directivity pattern. For instance, a human mouth radiates more energy to the front of the speaker. The radiation is attenuated and low-pass filtered when going to the backside of the speaker because of the shadowing effect of the head.

Most musical instruments have complex radiation patterns. They are caused by different modes of vibration in the bodies of instruments (e.g. in the string instruments) or multiple audio output points of an instrument (e.g. in the wind instruments). Another kind of factor in instrument directivity is the shadowing effect of the instrument player himself or herself.

Source directivity can be modeled in two ways: with directivity filtering or with a set of elementary sources. With directivity filtering, a point source is filtered with a filter whose parameters depend on the direction wherefrom the source is observed. Usually, with real-time applications, low-order filters that are symmetric to the main-axis of the instrument suffice. In the case of a set of elementary sources, multiple point sources are used instead of one to create the required directivity pattern.

3.2 Room modeling

3.2.1 Methods

There are three different approaches to model room acoustics computationally: wave-based, ray-based, and statistical modeling techniques. A classification of the methods is illustrated in the Figure 4 according to [Savioja 1999].

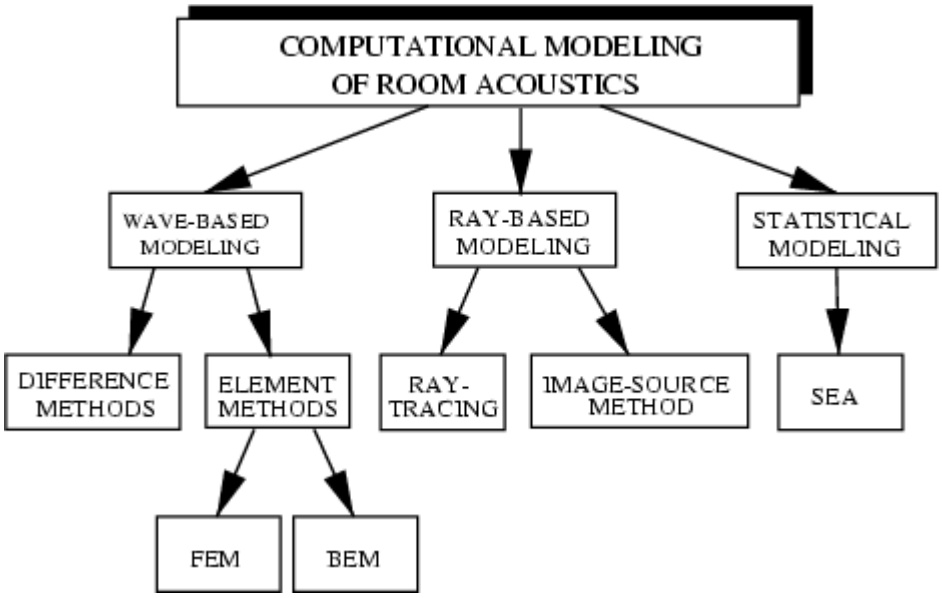

Figure 4. Principal computational models of room acoustics according [Savioja 1999]. Printed with a permission of the original author.

Statistical modeling methods are mainly suitable for noise level estimations in cases when sound is transmitted via different structures. The temporal behavior of a sound field is typically not modeled making these methods inappropriate for auralization purposes of virtual acoustics.

Because sound propagation follows the wave equation, the wave-based modeling approach that approximates the equation yields the best results. The downside is that this method is computationally rather complex and not well suitable for real-time applications. The computational complexity narrows the use of wave-based methods to small sized rooms and lower frequencies only. Wave-based methods include finite element method (FEM), boundary element method (BEM), and finite-difference time-domain (FDTD). FEM and BEM are methods, which fill the modeled space with small units, elements. Usually, these elements are used to calculate the frequency domain responses of a given field. The difference between FEM and BEM is that FEM discretizes the whole region and BEM discretizes the region boundaries only. In comparison to usual frequency domain approach of FEM and BEM, the main principle in the FDTD is that derivatives in the wave equation are replaced by corresponding finite differences. The FDTD approach produces a more suitable time-domain impulse response of the modeled space.

The third approach is ray-based modeling. Ray-based methods derive from computer graphics and treat the sound acting as rays, like light. This approximation is valid when the wavelength is long compared to the roughness of the surface, and on the other hand, short compared to dimensions of the surfaces in the acoustical space under modeling. The most common ray-based methods are ray-tracing and image-source methods. The idea

behind both of them is to find reflection paths from the source to the listener; the distinction is the way paths are searched for.

The basic ray-tracing algorithm works so that many rays are cast from the sound source to different, usually equally distributed over a sphere, directions, the rays then reflect from the surfaces according to specular reflection[1], and finally some of the rays hit the listener. The rays hitting the listener mark the reflection paths. The listener is usually modeled as a sphere with a big-enough finite volume to make enough rays to hit it.

In image-source method, no rays are cast, but instead the reflections from surfaces are modeled by generating mirrored image sources of the real source behind the reflecting surface. The direct paths from the image sources happen to come from the same direction the actual reflected path would come. Also the distance equals to the real reflection path. In other words, there is an image source for every reflection path from the source and the direct path from these image sources is used for calculations of the room response instead of reflected paths. Images of the images correspond reflection paths of multiple reflections. One advantage in image-source method is that the movement of the listener does not cause the need to recalculate the image sources, just the actual reflection path has to be formed and checked that it is obstacle free.

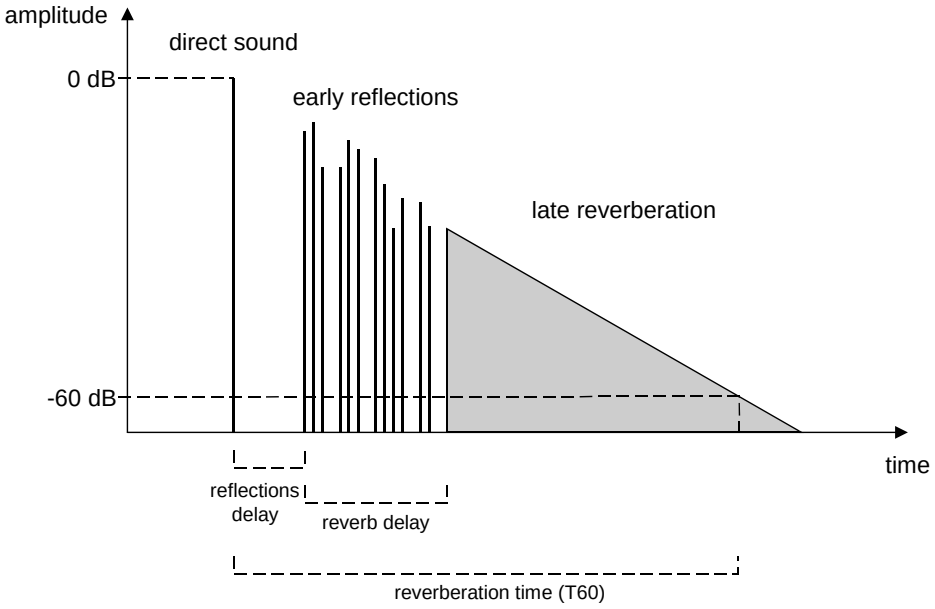

Figure 5. Room impulse response is typically divided into direct sound, early reflections, and late reverberation.

Figure 5 illustrates a typical division of the room impulse response: direct sound, early reflections, and late reverberation. Ray-based methods are efficient for finding early reflections, but become computationally heavy when later reflections are searched for. Usually in room simulations, the early reflections are calculated with these methods, but late reverberation is simulated with an efficient approximative recursive algorithm instead. One important class of such algorithms is feedback delay networks (FDN) that was introduced by Stautner and Puckette in [Stautner, Puckette 1982].

[1] In specular reflection, the angle of incident of an incoming ray is the same as the angle of reflection of the outgoing ray.

A feedback delay network (Figure 6) consists of multiple delay lines that are interconnected with a feedback matrix to form a feedback loop. The delay lines have unequal lengths. For instance, mutually coprime numbers can be used as lengths to minimize the collision of echoes in the impulse response. H_n filters at the end of each delay line simulate the absorption caused by the reflection and the distance attenuation. [Zölzer 2002]

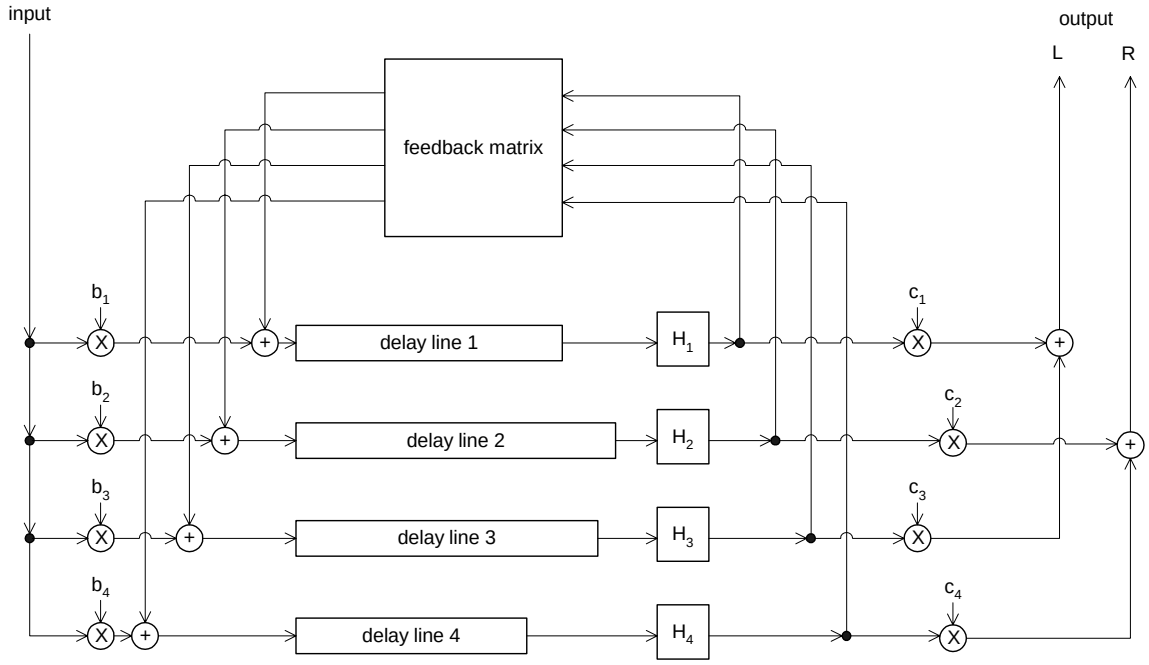

Figure 6. A feedback delay network.

3.2.2 Attenuation and absorptions

The distance attenuation of the sound follows the 1/r-law. It means that the sound pressure level drops 6 dB when the distance doubles. This has to be taken into account for all the paths from the source to the listener gained from the image-source or ray-tracing methods.

In addition to the 1/r-law distance attenuation, also another effect is distance dependent: air absorption. It also depends on the temperature and the humidity. The air absorption affects the sound by low-pass filtering. Air absorption and distance attenuation are usually implemented as a common filter, because they both are functions of distance.

Furthermore, on the reflection paths there is one more important absorpting phenomenon: the absorption caused by reflections from the surfaces. It is the most complex of all, because it is a function of many parameters: incident angle, the scattering and diffraction phenomena etc.

3.3 Listener modeling

When the direction of the sound source (image or real) is known, the source's sound is filtered with the HRTF of that direction and as an output two signals are produced; one for each ear. When listening to this signal, the source sounds like being in that direction.

As mentioned earlier, HRTFs are of individual nature, so they have to be measured somehow. This usually happens in an anechoic chamber by placing small microphones in the ear canals of the person and then measuring transfer functions from loudspeakers placed around the person to the microphones. Usually, there is a row of speakers that is moved by a turntable around the person and then the responses are measured from each speaker in each position of the turntable. Another option is that the position of the speakers can be fixed and the person sits on a turntable and spins with it. Figure 7 illustrates the former way of measuring HRTFs.

Figure 7. An apparatus for HRTF measurements in the anechoic chamber of the Institute of Sound and Vibration Research (ISVR) at the University of Southampton, UK. <URL: http://www.isvr.soton.ac.uk/FDAG/VAP/>. Printed with a permission of ISVR.

Naturally, it is possible to measure the responses only from a discrete amount of directions. When a sound is wanted to be reproduced coming between the measured directions, it is either just estimated by choosing the nearest measured direction or it can be interpolated between measured directions.

A straightforward approach is to implement HRTFs as FIRs. They can, for instance, be FIRs with minimum-phase reconstruction, and the ITD is then modeled separately for each ear. Frequency warping can also be used to make the filters computationally more efficient.

3.4 3-D sound reproduction

After calculating the required sound field for virtual acoustics, it has to be somehow created to the listener's ear canal. There are different ways.

3.4.1 Headphone reproduction

A natural approach is to use headphones. Playing the calculated sound data via good quality headphones works quite well. Even better results can be obtained by first equalizing the response by headphone specific compensation filter. One thing that can cause the virtual acoustical world to collapse is the movement of the head; the virtual world then spins with the head movement. This can be prevented with the use of head-tracker and then compensating the head movements by altering the orientation of the virtual listener accordingly.

3.4.2 Loudspeaker reproduction

Another approach is to use loudspeakers. Here the biggest problem is the leaking of the sound from the left channel to the right ear and vice versa. This so-called crosstalk has to be cancelled with crosstalk cancellation filters. The filtering works correctly only in a certain position between the speakers; this so called "sweet spot" is not wide. This limits the movement of the listener dramatically compared to other reproduction methods.

3.4.3 Multichannel reproduction

Also, multiple speakers around the listener can be used, but then we are not talking anymore about binaural reproduction. The disadvantages of multichannel reproduction are the limitation of virtual source directions to the directions of the speakers installed and the big amount of hardware required. The advantages, on the other hand, are that no listener modeling is needed and the virtual world follows automatically the turnings of the listener.

3.5 A typical DSP implementation

One way to implement the virtual acoustics in a resource-limited mobile device is illustrated in Figure 8. The direct sound is first attenuated with 1/r-law and then filtered with some HRTF approximation. The reflected sound, on the other hand, is first filtered with a room filter that tries to approximate the distance attenuation, the air absorption, and the attenuation on the reflections. It is typically a low-pass filter. Then, a delay line is used to generate early reflections and a recursive algorithm generates late reverberation. Finally, the direct sound and the reflected sound are added together to form the output for headphones. In this model, only the direct sound is HRTF-filtered.

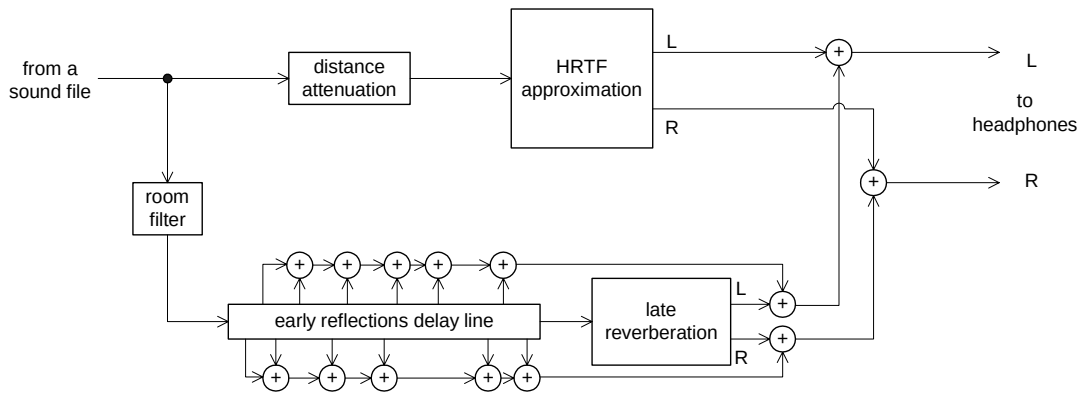

Figure 8. DSP blocks of a typical virtual acoustics implementation.

4. Existing 3D audio APIs

In this chapter, the most important already existing 3D application programming interfaces (API) that contain advanced audio processing features are introduced. These include Java 3D API and MPEG-4. Also VRML and X3D have possibilities for spatialized audio, but MPEG-4 extends their way of audio presentation and thus VRML and X3D are omitted here. Although, when describing MPEG-4, similarities to VRML are emphasized. Furthermore, 3D audio extensions to SMIL have also been studied [Siemens 2002], but they are not publicly available yet.

Two soundcard manufacturer's APIs are presented as well. They are Creative Lab's EAX and Aureal's A3D. DirectX audio, being the basis of EAX and also otherwise popular API in the Microsoft Windows world, deserves an own subchapter too.

The presented APIs are compared in section 6.5. Also the comparison against the new API this thesis introduces is there.

4.1 Java 3D API

Java 3D API is an extension to J2SE that provides tools to construct three-dimensional graphics for Java applications and applets. This chapter gives on overview of the API in general and then 3D sound part of it is described in more detail. [Walsh, Gehringer 2002] acts as a reference together with the API documentation [JAVA3D 2002].

4.1.1 High-level

Java 3D API is a high-level API that makes it possible for the programmer to concentrate on what to draw instead of how to draw. This distinguishes it from low-level rendering APIs like OpenGL and Direct3D. Usually, Java 3D API implementations are layered on top of these low-level APIs. Being high-level, Java 3D still offers also control over low-level rendering details if necessary.

4.1.2 Concept of scene graph

Java 3D is based on treelike structure to describe the three-dimensional world. Similar structure is used in many other 3D APIs and is called as a scene graph.

Scene graph stores the information of the scene in a tree that consists of nodes. Nodes can either represent objects or properties of the virtual world or they can be group nodes that contain other nodes. Group nodes organize the tree so that individual objects form together some bigger entity and those bigger entities can again be grouped together to form a higher level entity and so on. The result is a tree where leaf nodes represent the simplest

objects in the world and the world is represented by the whole tree together. Figure 9 illustrates one simple scene graph.

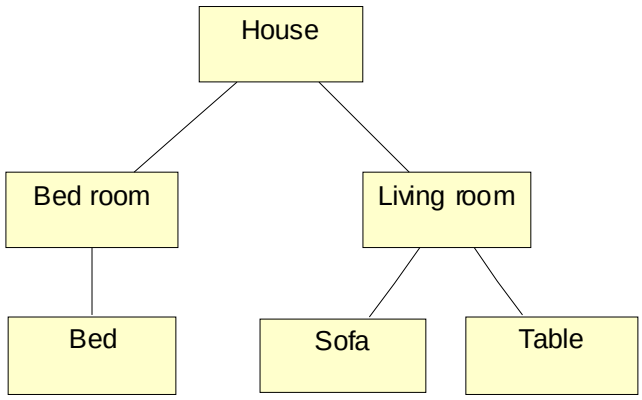

Figure 9. An example of a scene graph.

As Java is an object-oriented language, nodes in Java 3D are actually objects, instances of one of the class Node's subclasses. Nodes of the scene graph can be divided to four categories: shape nodes, environment nodes, group nodes, and to the ViewPlatform that forms a separate category alone. The shape nodes represent some geometrical visible 3D objects in the world such as cubes, points, or even something more complex like a teapot. The environment nodes are not (at least directly) visible. They affect the environment in an area of the virtual world. Light, Sound, and Fog are examples of environment nodes. The **ViewPlatform** node is a special node. It controls the position, orientation and scale of the virtual observer in the world. The scene is being viewed from the place of ViewPlatform.

Nodes can be divided further into smaller parts. **NodeComponents** are pieces of nodes and they hold properties or data of the node or nodes associated to them. For instance, the shape nodes have NodeComponents called Geometry and Appearance; former defining the geometry and topology of the shape and the latter defining among others the material the shape is made of. NodeComponents can be shared between multiple nodes. This makes it possible to, for instance, for the chair and the sofa to easily share the same Appearance, for example, black leather. There is no need to define Appearance separately for each shape.

4.1.3 Sound sources in Java 3D

The audio properties of the virtual world are defined in Java 3D with two different types of environmental nodes: Sound and SoundScape.

Sound is the base class for different types of sounds and provides methods for controlling the overall gain, muting, and looping of the sound, and setting the **MediaContainer** that stores the actual sound data to be played. Typically, MediaContainer is set up to read the data from a sound file, such as ".wav", located with an URL.

Sound class has various subclasses. A **BackgroundSound** defines an unattenuated, non-spatialized sound source that has no position or direction. It is useful for playing mono or stereo music track or an ambient sound.

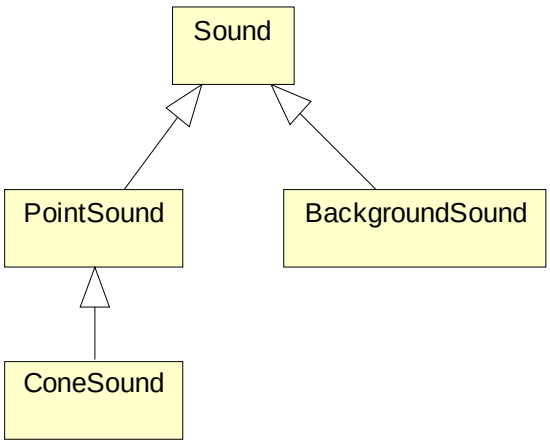

Figure 10. UML class diagram of Sound with its subclasses.

The other subclass of Sound is **PointSound** that defines a spatially located sound source. PointSound is an omnidirectional source that radiates equally to all directions. PointSound has methods for setting its location and distance attenuation curve. The attenuation curve is defined in an array of distance and gain scale factor pairs. Between defined distances the gain scale factor is linearly interpolated.

PointSound has a subclass **ConeSound** that is like a PointSound, but has directivity. ConeSound has definable direction and directivity pattern that tells how the sound of the source attenuates when listening from off the front direction of the source. The directivity pattern can be defined using three arrays of equal length: angular distance from the front direction of the source, gain factor, and low-pass filter cutoff frequency. This makes it possible for not only the gain to attenuate when moving from the front of the source towards the side of it, but also the sound to be filtered so that sounds with higher pitch attenuate more than lower sounds. The directivity is symmetrical around the main axis of the source.

4.1.4 Virtual acoustical environment in Java 3D

Class Sound is used to define sound sources, but also the other attributes of the sound can be controlled in Java 3D. Node **SoundScape** defines the space where the sounds are listened. SoundScape has two important fields: **AuralAttributes** that has various parameters to define the environment where the Sounds are listened, and **Bounds** that defines the space in the virtual world where the AuralAttributes are applied. A scene graph can contain multiple SoundScapes and this makes it possible for the acoustical environment to change when moving around in the world.

AuralAttributes is a sub-class of NodeComponent and defines the SoundScape. The settings in AuralAttributes include simple gain scale factor, sound velocity scale factor, and parameters controlling reverberation, distance frequency filtering, and velocity-based Doppler effect.

Java 3D's reverberation model is tunable with multiple parameters. It is divided into two components: to the early, distinct reflections and to the late reverberation. Both components have the separate settings for the gain and the delay time for the first reflected sound to reach the listener. The decay time, the echo dispersement (echo density), and the low-pass filter cutoff frequency of the late reverberation are definable, as well as the modal reverb density, which defines how smoothly the reverberation decays. An

alternative way to define late reverberation delay and decay is also offered: one can just define bounds of the virtual physical space (room); then delay and decay are calculated automatically internally. To complete the set of the reverberation parameters offered by Java 3D there is also one parameter that can be used to control the rendering: reverb order can be set to limit the number of reflections calculated and thus make the rendering less demanding.

Distance filtering can be used to simulate the air absorption, which has characteristics to attenuate higher frequencies more than lower ones. Distance filtering in Java 3D is defined giving distance/frequency pairs, which define the cutoff frequency of the low-pass filter applied for specified distances from the source.

The Doppler effect means the frequency shift sensed by the observer when the distance to the source changes. When the source moves towards the observer the observed frequency raises and the other way around when the distance gets longer. Using the Doppler effect, the sense of sound source movement gets stronger. The strength of the Doppler effect can be tuned with two scaling factors: one for the relative velocity of the sound source and one for the frequency shift caused by the velocity. If the Doppler effect is turned off, the frequency shift factor can be used to directly tune the pitch of the sound.

4.1.5 Physical real acoustical environment in Java 3D

In addition to the scene graph, Java 3D has also class **PhysicalEnvironment** that controls the physical environment where the view of the virtual world is generated. PhysicalEnvironment has accessors to **AudioDevice** that has ways to access physical parameters like distance and angle from the listener to the nearest speaker and the info if the headphones are used instead of speakers.

4.2 MPEG-4

Moving Picture Experts Group (MPEG) is one of the many Working Groups in the ISO/IEC. MPEG has been responsible of developing successful standards MPEG-1 and MPEG-2. After them, it continued its work by developing the MPEG-4 standard. The first version of MPEG-4 became International Standard in 1999. Since that, MPEG has worked to produce successive versions and addendums to MPEG-4, and furthermore, two successive standardization projects have followed, namely MPEG-7 and MPEG-21, initiated in 1996 and 2000, respectively.

In this chapter, overview of MPEG-4 is given with emphasis in the audio properties MPEG-4 provides. [Pereira, Ebrahimi 2002] and [MPEG] act as references in this chapter.

4.2.1 Coding of Audio-Visual objects

The scope of both MPEG-1 and MPEG-2 has been effectively compressing and then transferring audio-visual streams that contain natural audio and video content from a place to another. The content is mixed together in production phase and the consumption is of passive nature, spectating precomposed media. The MPEG-4 takes a different approach: coding of audiovisual objects.

The content in MPEG-4 consists of multiple audiovisual objects. These audiovisual objects can be of different nature: they can be such as arbitrarily shaped video objects, multichannel audio objects, or objects that contain only speech. The objects of MPEG-4

presentation are combined in the receiving end to form an audiovisual scene. The rules how to combine the objects are described using a scene description language that in the case of MPEG-4 is BInary Format for Scenes (BIFS). It is described in the later chapters. Forming scene in the receiving end makes it possible to make the composing procedure interactive and to enable for the user to interact with the scene. The scene can, for instance, be a 3D world that the user can observe from different viewpoints.

MPEG-4 also utilizes the concept of synthetic-natural hybrid coding (SNHC). SNHC means that the audiovisual object can be either natural, like video produced with a video camera or audio produced with a microphone, or synthetic, such as animation produced using computer graphics, music produced using synthesizer, or speech produced using speech synthesis. Natural objects can be transmitted within MPEG-4 stream in a traditional way, but synthetic objects can be produced in the receiving end using just the parameters transferred with MPEG-4 stream, thus saving bandwidth. At the end, both types of objects are mixed together to form a composed scene.

In addition to the bandwidth saved by transferring just parameters of synthetic objects, also the separate coding of natural objects saves bandwidth in general: the most optimal compression algorithm can be chosen for different kinds of natural objects. For example, speech can be coded with a speech codec and background music with a generic audio codec. This approach gives the most efficient kind of coding for both forms of audio and probably coding both combined with generic audio codec still maintaining the quality of speech would waste more bandwidth than two separate codecs together. Furthermore, different bit and sampling rates can be used for different sound components on the scene.

4.2.2 Structured Audio

In MPEG-4, the audio coding tools are divided into two major categories following the SNHC approach presented in the previous chapter. The division is illustrated in Figure 11. There are separate tools for the natural audio and for the synthetic audio. The natural audio tools are generic audio and speech, and synthetic audio tools are structured audio (SA) and text-to-speech interface (TTSI). SA is introduced in more detail in this chapter.

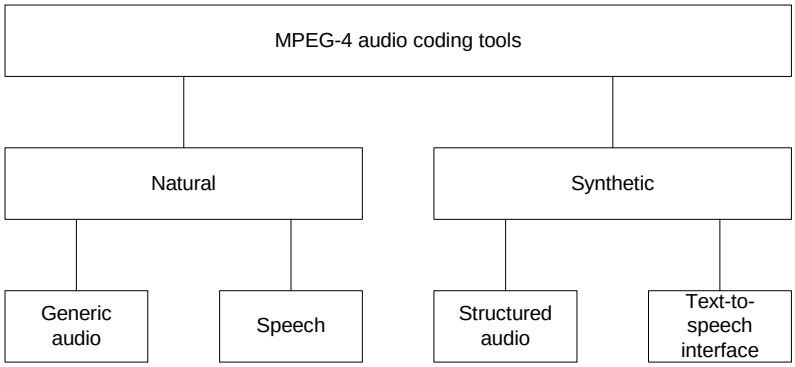

Figure 11. The division of MPEG-4 audio coding tools.

Structured audio (SA) is a concept of MPEG-4 that allows a very low bit-rate transmission of synthetic audio, such as synthetic music or sound effects. SA offers ways to describe the audio, not in a traditional form of sampled waveforms, but instead as the algorithms used to produce the audio signal with the necessary timing information. MPEG-4 defines two languages for these purposes: Structured Audio Orchestra Language (SAOL) for describing the algorithms and Structured Audio Score Language (SASL) for the timing and controlling of the algorithms. As a rough analogy with the real world could be that

SAOL describes how various instruments in the symphony orchestra sound and SASL describes the score the orchestra is playing. In addition to the instruments described as algorithms with SAOL, there is also a format to transfer sample data, Structured Audio Sample Bank Format (SASBF). SASBF data can be utilized in SAOL algorithms or used as-is to form wavetable based instruments. Figure 12 visualizes the structure of the SA decoder.

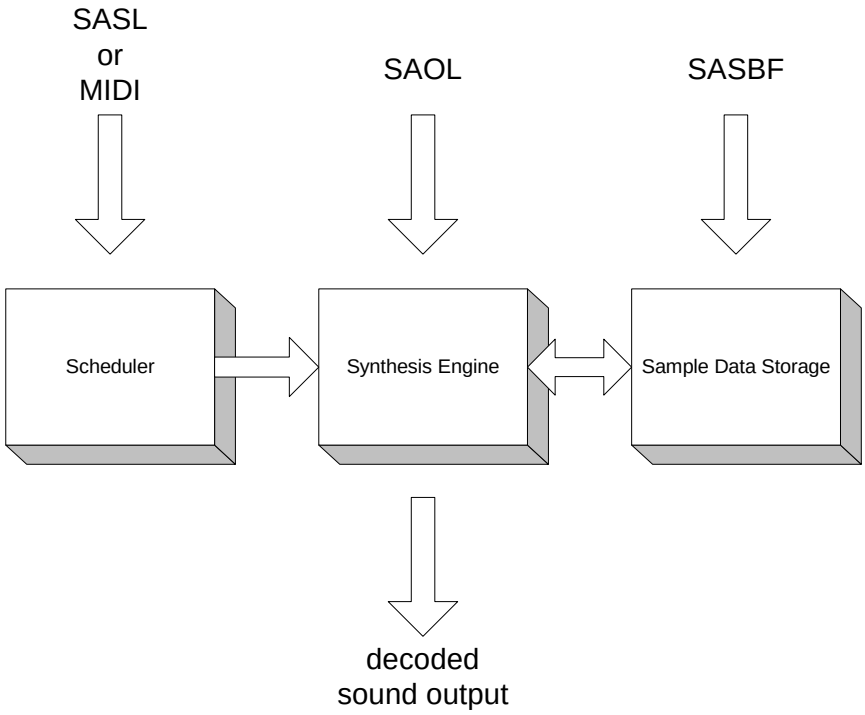

Figure 12. SA decoder.

SAOL is a programming language with C-like syntax and basic operators. The instruments are described in SAOL as networks of various digital signal-processing (DSP) routines. SAOL has a library of around 100 core opcodes that represent various kinds of mathematical and DSP functions, such as, signal generators, DSP filters, pitch converters, and delay functions. All the SAOL functions are normatively defined in the standard. This guarantees that the same SAOL stream always produces the same sound output when played with a MPEG-4 compatible player.

SASL controls the SAOL instruments. SASL contains the timing information to instantiate the instruments, to control their parameters, and to change the tempo. SASL events are sent to the scheduler part of the SA decoder. SASL events contain time stamps and the scheduler takes care of ordering the events to the correct order and then triggering them at specified times during the decoding process. Instead of SASL, SA decoder can also use MIDI stream as a score language.

4.2.3 BIFS

As stated earlier, MPEG-4 has a scene description language called BInary Format for Scenes (BIFS) for describing how to combine the various audio-visual objects. BIFS is strongly based on the Virtual Reality Modeling Language (VRML) [VRML97] and its successor Extensible 3D (X3D). Compared to the combination of Java and Java 3D presented earlier, BIFS is not a full scale powerful programming language, but as it still defines runtime semantics, it is not pure 3D file format either. It is something between.

Runtime dynamic behavior of BIFS can be defined using ECMA Script that is a language similar to commonly known Javascript.

BIFS describes the scene using a scene graph concept similar to Java 3D. The concept of scene graph was described in the Java 3D chapter. The scene described with BIFS can be either flat or 3D, and contain many kinds of audiovisual objects. There can be visual objects such as rectangular video, video with shape, synthetic face and body, generic 3D objects, text, and graphics; and audio objects, such as speech, natural music, synthetic music and information about room effect. Next subchapters tell more about audio nodes in BIFS.

4.2.4 AudioBIFS

Audio nodes of BIFS are called as AudioBIFS. They were specified in two stages: first 10 AudioBIFS nodes came out with the first version of MPEG-4 and were later followed by four additional nodes defined in the first amendment of MPEG-4 Systems standard. These latter nodes are called as Advanced AudioBIFS and address mainly to 3D sound propagation in room. In this subsection, the core AudioBIFS nodes are described. The following subsection covers Advanced AudioBIFS. All audio nodes have been grouped in this thesis to five categories, and this grouping can be seen in Figure 13.

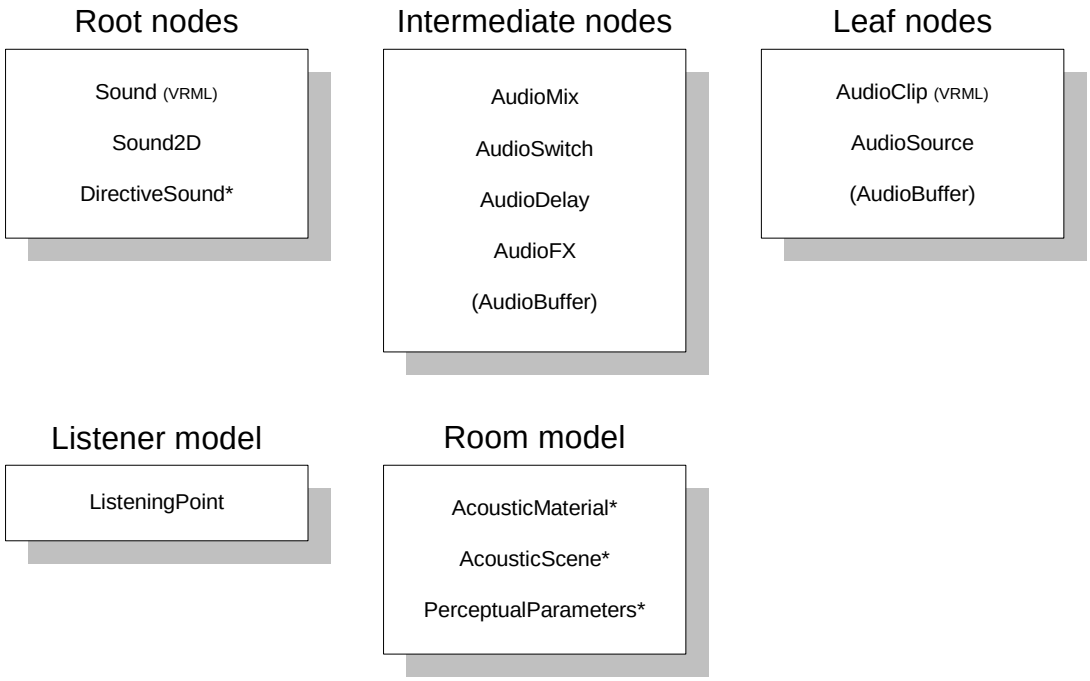

Figure 13. AudioBIFS and Advanced AudioBIFS nodes. Advanced AudioBIFS nodes are marked with asterisk (*). AudioBuffer is stated twice because of its dual nature.

Only two of the AudioBIFS nodes are inherited from VRML: Sound and AudioClip. **Sound** is the node that attaches the sound source into the BIFS scene. It has a location and direction in the 3D space. A Sound has also four fields to define direction and location dependent sound attenuation. This directivity is specified with two distances both to the back and to the front of the source. They define two ellipsoids that are symmetric to the main axis of the source. Inside the inner ellipsoid the observed sound pressure level stays constant; between the surfaces of the two ellipsoids level decreases 20 dB when the distance grows; and outside of the outer ellipsoid the source is silent to the observer. A

Sound has also a Boolean field spatialize that can be used to toggle the spatialization of the field on or off. Spatialization means rendering the direction and the location of the source to the sound heard by the user.

Sound node has also a field for a source. This source can, for instance, be an **AudioClip**. An AudioClip associates Sound to the actual source of sound specified with Universal Resource Locator (URL). An AudioClip has fields to control the starting and stopping of the playback in specified times. Also looping is possible. With an AudioClip node it is possible to include only sound that is an interactively downloaded audio clip; streaming is not possible. MPEG-4 adds a new parallel node to this VRML originated AudioClip; an **AudioSource** is capable to play also audio streams made with MPEG-4 audio coding tools described earlier.

Like AudioClip, also Sound has MPEG-4 specific dual nodes: Sound2D and DirectiveSound. These three are called here as root nodes, because they are the nodes that are associated to the scene graph and other audio nodes associated to them can be considered to form an audio sub-tree of that specific sound source in the scene. Basically an audio sub-tree in the scene graph can be understood so that leaf nodes produce sound to the audio sub-tree, intermediate nodes form the signal processing network that connects the leafs to the root node that acts as a virtual source in the scene. The concept of the audio sub-tree is described in more detail later.

As being Advanced AudioBIFS node, DirectiveSound of root nodes is described in the next subsection, but **Sound2D** is described here. Sound defines the location in 3D space, which is sometimes unnecessary for 2D applications. Sound2D addresses to this need and defines the place of the sound source on a plane with two coordinates instead of three. The vertical plane has a width of 2 meters and is 1.5 meters high. The viewpoint is 1 meter away right in front of the plane facing its center point. Sound2D does not have fields for defining the sources directivity pattern, but the spatialization can be toggled.

The intermediate nodes of the audio sub-tree form the signal paths from the leaf nodes to the root node. They mix the sound of their children (that are leaf nodes or other intermediate nodes) and process it in some way and output sound to their parent node in the tree. The tree-like network makes it possible for one virtual sound source in the scene to consist of multiple sound streams (leaf nodes) that are processed and mixed in a certain manner. The intermediate nodes are AudioMix, AudioSwitch, AudioDelay, AudioFX, and to some extent AudioBuffer that has also leaf node behavior.

AudioMix and **AudioSwitch** serve the same need: to mix multiple sources (children) to the output (parent), but they have different kind of ways to control the mixing. AudioMix has a matrix that defines the proportions of the input channels that are mixed to the output channels. AudioSwitch is simpler: no gains can be specified; the inputs connected to the outputs are defined with boolean values.

AudioDelay delays its children's outputs with the same, specified amount of time. It can be used to fine-tune the synchronization between various streams on the scene.

AudioFX performs arbitrary audio filtering to its children and passes the filtered audio to its parent. AudioFX has a field that specifies the filtering algorithm in SAOL of structure audio tool described earlier. Also SASL can be used additionally in a separate field to control the parameters of the filtering algorithms if necessary. AudioFX is a handy tool to add for instance reverberation to anechoic speech generated by text-to-speech synthesis.

AudioBuffer has a dual nature: it acts like AudioClip, producing sound like a leaf node, but on the other hand, it can be used to capture sound from its children as well. This captured sound is the sound AudioBuffer outputs. AudioBuffer has also functionality to enable transferring MPEG-4 encoded sounds to be used as wavetables of the SA decoder.

In BIFS, the viewpoint to the scene is specified with Viewpoint node. The point where the sounds are heard is usually the same point. However, the observation point of the sounds can also be specified separately with a **ListeningPoint** node. This makes the listening point independent of the viewpoint, if necessary.

4.2.5 Advanced AudioBIFS

Advanced AudioBIFS add 3D sound propagation properties to original AudioBIFS. Advanced AudioBIFS consists of four new nodes: DirectiveSound to replace Sound when spatial properties of the sound source are essential; AcousticMaterial and AcousticScene to define acoustical properties of the space (room effect) using physical parameters; and PerceptualParameters to define the same properties in alternative way, using perceptual parameters.

DirectiveSound has the same kind of semantics than Sound, but adds some features. The setting of the source directivity is more fine-grained. Arbitrary numbers of angles can be specified; and for all of those angles, parameters directivity and frequency can be given to define frequency-dependent directional filtering. Furthermore, the speed of sound can be changed to affect the propagation delay, distance-dependent attenuation can be defined by setting the distance where attenuation is 60 dB, and specific air absorption filter can be switched on. Air absorption is simulated according to distance-dependent air absorption curves defined in ISO9613 standard. DirectiveSound has also a Boolean field to turn the room processing specified by the nodes presented next, completely off.

AcousticScene can be used to define a smaller region of the entire scene to limit the acoustical processing in that area only. Only the surfaces on that area will by modeled when the ListeningPoint is inside. With AudioScene, also the late reverberation parameters are defined: frequency dependent reverberation time (T60), delay after the late reverberation starts, and the level of the reverberation can be defined.

Material node gives the visual properties of a geometrical shape object on the scene. Like Material node, **AcousticMaterial** can be used to define visual properties of object, but it has also additional fields that define the acoustical properties of that same geometrical object; an object can have frequency dependent transfer functions for both reflections from its surfaces and for transmission of sound through the object.

As told earlier, **PerceptualParameters** can be used as an alternative way to AcousticScene and AcousticMaterial to specify how rooms and objects in those affect the observed sound. With AcousticScene and AcousticMaterial, the space was specified using physical measures. With PerceptualParameters, perceptual measures are used instead. PerceptualParameters is given as a field of a DirectiveSound node, thus it can be specific to each virtual source. The given perceptual measures control the energy division on various parts of the room impulse response (RIR) in both time and frequency domains. RIR is divided into four parts in the time domain: direct sound, directional early reflections, diffuse early reflections, and late reverberation; and into three parts in the frequency domain: low, mid, and high frequencies. These parts are specified by the user giving appropriate time limits and frequency borders. The effects that various perceptional measures given have to the different parts of RIR are described in Table 1. In the table, the

perceptual parameter is on the z-axis and the different frequency/time bands are on the x-axis. There is a field refDistance that defines the distance where parameters apply; when the distance changes the parameters are scaled accordingly automatically.

Table 1. The effect of various perceptual parameters on energy (E) and on decay time (T).

	DIRECT SOUND			**DIRECTIONAL EARLY REFLECTIONS**			**DIFFUSE EARLY REFLECTIONS**			**LATE REVERBE-RATION**		
frequency band	**L**	**M**	**H**	**L**	**M**	**H**	**L**	**M**	**H**	**L**	**M**	**H**
source-Presence	E											
source-Warmth	E											
source-Brilliance			E									
room-Presence										E		
late-Reverberance											T	
heavyness										T		
liveness												T
envelopment	E			E								
directFilter-Gains	E	E	E									
inputFilter-Gains	E	E	E	E	E	E	E	E	E	E	E	E
omniDirectivity				E	E	E	E	E	E	E	E	E

Fields sourcePresence, sourceWarmth, and sourceBrilliance affect the direct sound. The late reverberation is affected by roomPresence, giving the overall gain of it, and lateReverberance, heaviness, and liveness, specifying different decay times. Envelopment specifies the relative energies of the direct sound and the directional early reflections.

Furthermore, three additional filters are specified with the fields directFilterGains, inputFilterGains, and omniDirectivity. They all are specified in a similar way, using three figures: gains for low, mid, and high frequencies. DirectFilterGains sets a filter for the direct sound only, making an occlusion[2] effect possible. OmniDirectivity sets filters for the room part of the RIR only affecting thus directional early reflections, diffuse early reflections, and late reverberation. inputFilterGains affects all parts equally.

[2] The occlusion effect means the shadowing of the sound by an object between the source and the listener.

Additionally, out of the table, modal density of the RIR can be specified. It controls the density of the resonances in the frequency-domain response. Early decay time can be specified with field runningReverberance by giving time when the level decreases 10 dB.

4.3 A3D

A3D is a positional sound C++ API and engine made by Aureal Inc. The latest version of A3D API is 3.0 [A3D] and it is presented in this chapter. A3D 3.0 never came popular; Aureal had economical problems and Creative Labs became the owner of the A3D technology. The predecessors, A3D 1.0 and A3D 2.0 are popular APIs, though. The main differences between versions are that A3D 2.0 adds rendering of early reflections to the basic positional audio that A3D 1.0 offers, and A3D 3.0 adds support for late reverberation, sources with finite volume instead of point sources, and the possibility to use MPEG audio layer 3 compressed audio sources. [Hagen, Muschett 2002]

A3D API 3.0 follows Component Object Model (COM) [COM] software architecture developed by Microsoft and is implemented as a COM server. The A3D 3.0 server is accessible for application developers via various interfaces that follow COM model. The following sections describe these interfaces with their most important functionality.

4.3.1 IA3d5

IA3d5 is the top-level interface of A3D and all the other interfaces are created or queried from it. The initialization and finalization of the audio engine is done via methods IA3d5 offers.

IA3d5 hosts the frame buffer based architecture of A3D. When the audio scene is described using the various methods of A3D, the scene information is not passed to the audio rendering hardware or software immediately. Instead, the scene description data is accumulated first to a frame buffer and after application has completed the scene, it calls IA3d5's Flush method to take the accumulated audio frame into use to the rendering engine. This is similar to the concept of double buffering in computer graphics. Instead of graphical information, an audio frame is a collection of acoustical parameters describing how the scene should be rendered. These parameters include, for instance, updates to listener and source positions, and dynamically changing obstacles in the scene with their geometry and material. In other words, the frame buffer is first cleared with Clear method, then the updates to the virtual audio world are described, and finally these updates are sent to the renderer with Flush method.

IA3d5 has the global functionality of the virtual audio world. The distance attenuation model and Doppler effect can both be scaled with scaling factors from their natural settings. Setting both factors to 1.0 equals natural 1/distance attenuation and 340 m/s speed of sound for use in Doppler calculations. Global equalization is also possible, but it is limited into one fixed band treble attenuation only, of which mentioned use case is underwater acoustics simulation. Global gain is controllable, too.

Methods for tuning in various ways the computational capacity used for rendering are offered. The maximum delay for reflections can be set to limit the memory buffer needed for storing the waveform data to be heard as an echo later. A3D has also a concept called Resource Manager. Usually the underlying hardware has some maximum number of 3D audio sources and reflections that can be rendered simultaneously. The virtual world might have more audio sources active in a same time. To overcome this problem, A3D

introduces a real-time software rendering engine, namely A2D. If the audio hardware does not have enough capacity to render all the necessary audio sources, Resource Manager gives the less important source to A2D for rendering them using efficient software algorithms. If also software channels of A2D run out, the least important sources are converted as virtual sources. A virtual source is playing in means that the pointers to the audio data are updated as time passes, but it is not rendered at all and is therefore inaudible using only very limited amount of computing resources. IA3d5 offers methods to control the amount of software rendered A2D sources and as well it is possible to limit the amount of hardware sources used. To classify sources according to their importance, Resource Manager uses two values for each source, audibility and priority. Audibility is calculated from distance attenuation, gain, equalization, and occlusion of the specific source. The priority of each source can be set by the application. The weight how much priority counts compared to audibility for Resource Manager can be set.

The used coordinate system can be chosen between left-handed and right-handed versions. Also geometrical scaling is available making possible to specify how many units correspond to one meter in the real world. This enables use of arbitrary unit for length and makes audio scene geometry calculations easy to cooperate with visual calculations – what ever their scale is.

The reproduction method can be accessed. There are supported modes for headphone listening and for two different loudspeaker settings: speakers set into narrow angle and to wide angle. In the case of four output channels, they can be grouped as two stereo channels or as one quad channel with rear speakers.

4.3.2 IA3dListener

Interface IA3dListener provides controls for listener model. IA3dListener has methods for setting the position, orientation, and velocity of the listener. Velocity is used for Doppler calculations.

Reference to IA3dListener is gained by querying from IA3d5.

4.3.3 IA3dSource2

IA3dSource2 controls the source model of A3D. It has the same methods as IA3dListener to set the position, orientation, and velocity, but otherwise it has richer functionality.

IA3dSource2 is created with NewSource method of IA3d5. NewSource is given a parameter that tells if the source is of 3D or native type. 3D sources are rendered using spatialization and environmental effects like reflections, occlusion and reverb; the first order reflections and occlusion calculations can be switched off later if necessary. The native sources, in the other hand, are played as are, using just gain and pan control.

There are two ways to associate audio data into a source. With straightforward method LoadFile you can load an audio file or you can use a sequence of the following methods: SetAudioFormat, to describe the audio format used; AllocateAudioData, to allocate memory and resources; Lock, to get a pointer to the part of audio data buffer where it is safe to write without affecting current playback; and after copying the audio data UnLock, to release the pointer of the audio buffer for playback.

For playback control there are methods Play and Stop, for starting and stopping the playback, respectively. Play command is accumulated into frame buffer and takes place after Flush, but unlike Play, Stop is signaled immediately. Play method takes as an

argument about if the playback will continue in a loop or is the data played just once. Playback position can be controlled with sample accurate accessors Set- and GetPlayPosition, to control it as bytes, or with accessors Set- and GetPlayTime, to control is as seconds.

As was mentioned before, IA3d5 has the global scaling factors for some parameters. These same parameters can be scaled source specifically as well. These scaling factors settable from IA3dSource2 include the distance attenuation model, Doppler effect, equalization (that is limited into one fixed band treble attenuation only), and gain. The distance attenuation model can furthermore be fine-tuned with methods Set- and GetMinMaxDistance. SetMinMaxDistance takes three arguments: min distance, max distance, and behavior after max distance. When the source is closer than min distance, the distance dependent attenuation is not in effect. With this feature, the sources can be made audible in a larger area than just applying strictly natural distance attenuation model. Max distance specifies the distance where to the attenuation increases. Further than max distance, the behavior argument specifies if the attenuation either stays in a constant level it has reached in max distance or becomes muted after max distance.

The directivity of the source is accessible via methods Set- and GetCone. SetCone takes three arguments: two angles (inner and outer) and a gain factor for source radiation towards back of the source. The radiation pattern is calculated so that it is one when the angle from the source's main axis to the listener is below inner angle; when the angle is between inner and outer angle, gain is interpolated between one and backward gain; and when the angle is bigger than outer angle, the applied gain is constantly the backward gain. The source directivity is applied only for direct sound; for the reflections, it is ignored.

A3D 3.0 enables volumetric sources. That means that sources can have finite space instead of being point sources. With methods Set- and GetVolumetricBounds the dimensions of the source can be accessed and the rendering characteristics for volumetric sources are specified using method SetVolumetricDamping. It takes as a parameter a structure with various values. The occlusion of a volumetric source is calculated based on two values: size-relative damping fraction and visibility-relative damping fraction. The former is the relative size of the occluding polygon compared to the combined size of sound source and occluding polygon together. Visibility-relative fraction corresponds how many source polygon corner points are occluded; there is no straight-line form source polygon point to the listener. The weighting between the two damping factors can be set. The behavior outside the volumetric source is specified using parameter AzimuthPan that disables volumetric effect when set as 0 and on the other end, when set as 1.0 makes all the points on the surface of the source generate sound at full level maximizing the volumetric effect. Behavior inside the volumetric source can be switched either to mono, to make all the speakers give the same signal at full level, or to normal point source.

The pitch of the source can be scaled and if the sources are of native type, the stereo panning can also be accessed. The calculated audibility and the amount of occlusion can be queried if necessary for instance to make decision in the application program about possibly manually dropping hardly audible source. Priority of the source needed by the aforementioned Resource Manager can also be set to fine tune the automatic dropping mechanism.

The reflections of the sound can be scaled with two factors: the gain and delay scaling factors. Setting them to 1.0 correspond natural gain and delay for the reflections,

respectively. The interface IA3dGeom2 also offers methods for these two factors, but unlike case is here, they affect globally to all sources.

4.3.4 IA3dReverb

The reverberation settings in A3D are stored as IA3dReverb objects. New IA3dReverb objects can be created with NewReverb method of IA3d5 and it also has a method to chose which reverb object is in use. There can be multiple instances of IA3dReverb, but only one reverberation setting can be active in a time.

The reverb can be set either by using one of the 26 predefined named settings and fine-tuning it with three parameters or by setting 12 reverb parameters to create a fully custom reverb.

The presets can be fine-tuned with three parameters: decay time, damping factor, and volume. Decay time defines the T60 value, that is the time it takes for the reverberation to attenuate 60 dB. Damping factor controls how high frequencies damp compared to middle and low ones. The volume tells the level of the reverb effect compared to the direct sound.

To make a custom reverb, the following 12 parameters are needed. The level of the room effect is controlled with two parameters: one general and other relative for high frequencies. This room effect level is a general level for all room related effects and additionally separate relative levels can be set for reflections and reverb only. The decay time of the reverberation is also specified with two figures: decay time at low frequencies and then relative to it, at high frequencies. Delay that it takes for the first reflection to arrive after direct sound and delay from it to the late reverberation can be set. Also modal diffusion and echo density of the late reverberation can be set. Furthermore, for the values specific to high frequencies, the reference frequency can be set.

The amount of reverb (how wet or dry it is) can be set also individually for each source with a method of source: SetReverbMix; also the treble attenuation for the reverb can be set here if, for instance, wanted it to match similar settings of direct path.

4.3.5 IA3dGeom2 and IA3dList

To be able to calculate reflections and occlusions, the A3D engine needs information about the geometry of the scene. Usually, the geometry does not have to be so accurate as with visual applications, but simplified geometry is still needed. The geometry is defined with a IA3dGeom2 object. IA3dGeom2 is queried from IA3d5 like IA3dListener.

Like usually in 3D graphics, the geometry in A3D consists of polygons. All geometrical objects are defined using either polygons of three corner points, namely triangles, or with polygons having four corner points, namely quads. Those polygons define the surfaces of the object. They are the most primitive objects and therefore called as primitives.

In A3D, the primitives are described using a code block that starts with method Begin and ends with method End. The Begin takes as an argument which kind of polygons are used, triangles or quads. Between End and Begin, the corner points, also known as vertices are set calling method Vertex. When correct number of vertices has been set, the polygon is ended and the next primitive is started automatically even that Begin/End block has not ended yet. Multiple polygons of the same type can be defined inside one Begin/End block.

For the reflection calculations, knowing the normal of the surface is necessary. The incident and the reflected angles are calculated based on normal. The normals for each

polygon can be set manually before calling Vertex by calling Normal or if Normal is not called, the normals are calculated automatically. In A3D the surfaces are flat shaded, so each polygon can have just one normal direction.

Just like normal, each polygon can also have own material properties that are tied to polygon using BindMaterial. The materials are described in the next chapter.

In addition to basic primitives, triangles and quads, there are also primitives called subfaces. They take also the form of triangle or quad, but instead of being independent, they are placed onto surfaces. Their special property compared to basic primitives is that they make the surface they are lying in transparent. Subface triangles and quads are an efficient way to make openings to otherwise solid surfaces. The transparency of the subfaces can be tuned.

Various rendering features can be controlled from IA3dGeom2. Both occlusions and early reflections can be switched on or off separately for group of polygons. Also global control of them is possible. The occlusion and reflections calculations take remarkably processing power. The power consumption can be reduced by setting the intervals of the occlusion and reflection calculations longer. They can be set separately in A3D. Default value of the interval is one meaning that calculations are performed separately for each frame.

IA3dGeom2 also has various methods for using transformation matrices to transform objects and add hierarchy to the virtual world by grouping objects under the same matrix. Even listener or sources can be bound with a matrix if they are, for instance, wanted to move with an object. Transformation matrices are widely used in computer graphics. The details are omitted here.

A3D has also another way - other than transformation matrices - to group objects: lists. An object of IA3dList interface is first created with NewList of IA3dGeom2. IA3dList has Begin and End methods to define vertices in a similar manner as was the case with AS3dGeom2, but in addition, IA3dList has also Call method that actually sends the geometry to the engine. With lists, the geometry can be defined just once, but used multiple times, and thus computing power is saved.

4.3.6 IA3dMaterial

Different kinds of materials reflect and transmit sound in different ways. In A3D, each object can be rendered using different material. The materials are controlled with IA3dMatrial interface. The IA3dMaterials are created using NewMaterial of IA3dGeom2 and then after the material properties have been set, the material can be taken into use with method BindMaterial of IA3dGeom2. All the polygons sent to engine after that will have the specified material properties until next material is bound.

An IA3dMaterial has four parameters to be defined. It has reflectance to specify how much sound is reflected and transmittance to tell how much sound is transmitted when the sound has to travel via an obstacle. Both reflectance and transmittance have figures both to overall and high frequency specific attenuation.

4.4 DirectX audio

This subchapter describes the audio part of Microsoft DirectX family. Microsoft DirectX is a set of APIs for creating efficient multimedia applications for Microsoft Windows

platforms. There are versions for C/C++, C#, and Visual Basic languages. DirectX provides hardware abstraction layer (HAL) that hides device-specific dependencies of the hardware. The level of abstraction is chosen so that it is as close to hardware as possible still being so general that no code has to be rewritten when using different hardware. If DirectX based application is run on a hardware that does not support some feature used, DirectX might still support this specific feature in software, through so called hardware emulation layer (HEL). [Kovach 2000]

DirectX is divided into several APIs that support different fields needed in multimedia applications: input devices, graphics devices, and network connections among others. Here we concentrate on the audio features of DirectX version 9.0. They are divided into two APIs: DirectMusic and DirectSound. These two interfaces became more unified in the 8.0 version of DirectX: one major change from version 7.0 being that as DirectMusic was previously targeted for message-based musical data only, it now supports wave-based data as well. Manually parsing and streaming of the wave data directly to DirectSound is not necessary anymore, and wave files can instead be loaded and played by DirectMusic. DirectMusic can be thought to have a higher abstraction level than DirectSound. DirectMusic includes synthesizer to play, for instance, MIDI and then feed the generated wave data to DirectSound that is, on the other hand, specialized just into wave audio.

Like A3D, DirectX audio follows Microsoft's Component Object Model (COM). The most important interfaces of the DirectX audio 9.0 [DirectX 9.0 2002] are described in the subsequent sections. The features important to 3-D sound and virtual acoustics are emphasized. The most important interfaces are illustrated in Figure 14.

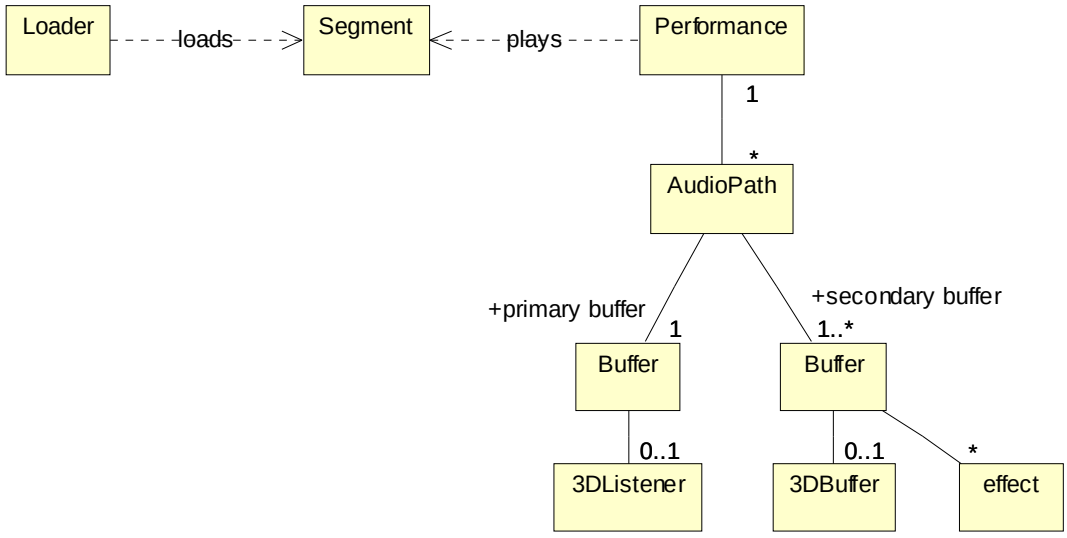

Figure 14. The most important classes of DirectX audio and their relations. Prefixes IDirectSound and IDirectMusic are omitted. effect can be any effect DirectX offers.

4.4.1 IDirectMusicLoader8

IDirectMusicLoader8 is an interface that is capable of loading different kinds of audio content from disc or from other resource, such as a memory location. Audio files can be among others MIDI files, WAV files, or DirectMusic Producer segment files.

Once a piece of audio data is loaded, it is typically represented in DirectMusic as an object that implements the IDirectMusicSegment8 interface described in the next section.

4.4.2 IDirectMusicSegment8

A segment in DirectMusic is the basic unit of playable data. Segment gives a common abstraction level, for example, to MIDI sequences and waveforms.

4.4.3 IDirectMusicPerformance8

The Performance is the object that manages the flow of data from the source to the synthesizer. The audio system is initialized with Performance's methods. Performance has also methods for starting and stopping the playback of an associated segment.

Performance uses AudioPath to playback segments. Performance has a default AudioPath associated to it, but other AudioPaths can be used as well. AudioPath is described in the next section.

4.4.4 IDirectMusicAudioPath8

AudioPath manages the flow of audio data from the performance to the final mixer. AudioPath can direct data via various DirectSound buffers before it enters to the primary DirectSound buffer where the final output is mixed. Buffers on the AudioPath can contain, for example, different musical effects or 3-D sound processing. There can be multiple AudioPaths so that different sound sources can have individual settings, such as 3-D location coordinates.

4.4.5 IDirectSoundBuffer8

DirectSound has two kinds of buffers: secondary and primary. Each sound source has one secondary buffer and each application has one primary buffer. Secondary buffers feed the primary buffer. Secondary buffers can have one or more effects associated into them.

Buffers have methods for controlling, for instance, playback level and rate. Effects described later are associated with buffers.

4.4.6 IDirectSound3DBuffer8

3DBuffer is an interface that can be asked from a secondary buffer when 3-D audio is used. Alternative way to get 3DBuffer is to query from the AudioPath. 3DBuffer has methods for setting various 3-D parameters for the sound source represented by that secondary buffer.

The location and the velocity of the sound source can be defined with methods of 3DBuffer. They are set using Cartesian vectors x, y, and z. The velocity definition affects only the Doppler effect and does not actually move the sound source. The movement has to be done by manually altering the location.

The sound source has also orientation that is used with its directivity calculations. The directivity of the source can be set using two angles: InsideConeAngle and OutsideConeAngle. Within InsideConeAngle the sound level is in its normal setting and outside of OutsideConeAngle the sound level is at specific level set with method SetConeOutsideVolume. Between the defined angles the sound level is interpolated between the normal level and the outside level. The directivity is symmetric with respect to the source's orientation axis.

The distance attenuation behavior of each 3-D sound source can be altered by setting two values: the minimum and the maximum distances. They specify the limits for the sound attenuation to take place. Closer than the minimum distance the sound does not get louder anymore when coming closer to the listener and when going farther than the maximum distance the attenuation does not increase anymore.

When the secondary buffer for 3-D audio processing is created, a 3-D algorithm can be specified. The choice affects only when processing is done on software using HEL. The alternatives are “no virtualization”, “hrtf light”, and “hrtf full”. The first one uses simply stereo panning and therefore the vertical axis is ignored for location. The second and the last one use 3-D audio processing with different qualities. The exact difference between the latter ones is not documented.

4.4.7 IDirectSound3DListener8

Like secondary buffers represented sound sources in the virtual acoustical world, the primary buffer represents the listener. Interface 3DListener can be queried from the primary buffer or alternatively from the AudioPath. 3DListener can be used to control spatial parameters of the listener and also general parameters of the acoustical environment.

The location and the velocity of the listener can be set with similar methods using Cartesian vectors like was the case with sound sources. There is only one exception: the orientation of the listener is defined using two vectors: front vector and top vector. With sound sources, only front vectors were needed, because sources are axis symmetrical.

The global parameters that can be controlled from the 3DListener are scaling factors for distance, rolloff, and Doppler. Distance factor scales the unit of distance used and makes it possible for using, for instance, feet instead of standard unit meters to express all the distances in the application. Rolloff factor can be used to scale the natural 1/r-attenuation of sound and Doppler factor to scale the amount of Doppler effect from its natural amount.

Every change to the settings of 3DBuffer or 3DListener causes recalculations to 3-D processing. DirectSound offers a way to group various changes together and commit them simultaneously. This is called as deferred settings. All the setting methods of 3DBuffer and 3DListener have a possibility to use flag DS3D_DEFERRED. Using this flag causes the settings not to affect immediately, but instead to accumulate. The accumulated settings can be then committed simultaneously by calling method CommitDeferredSettings of the 3DListener. Thus, the 3-D calculations have to be performed just once even if multiple parameters have to be changed.

4.4.8 Reverberations and other effects

Effects in DirectSound are set on secondary buffers. One secondary buffer can have multiple effects associated. DirectX has nine standard effects available, but also other effects can be registered into the system. The standard effects are chorus, compression, distortion, echo, environmental reverberation, flange, gargle, parametric equalizer, and Waves reverberation. All the effects have multiple parameters to control settings like dry/wet-ratio, modulation depth and frequency etc. The two reverberations are described now in more detail.

Waves reverb is based on the Waves MaxxVerb technology [MaxxVerb 2000]. Waves reverb has settings for input gain and reverb mix in decibels, for reverb time in

milliseconds, and for high-frequency ratio of the reverb time. Waves reverb is intended to be used with music.

The other reverberation, the environmental reverb of DirectX provides partial support for environmental reverberation defined in Interactive 3-D Audio Level 2 [I3DL2 1999] specification. The easiest way to use environmental reverb is to set parameters by using presets. Altogether 30 preset settings similar to I3DL2 are offered. Other way to use environmental reverb is to set all the reverberation parameters by hand. The parameters are the same described in I3DL2 listener property set and can be found from Table 2. Furthermore, there is also a parameter for setting the quality of the reverb meaning the compromise between the perceived quality of the reverb and the computing costs.

Table 2. The reverberation parameters of I3DL2 listener property set.

Parameter	**Description**
Room	Intensity of the room effect.
Room HF	Attenuation at high frequencies in the room effect.
Room rolloff factor	Scaling factor for the 1/r-attenuation of the room effect.
Decay time	Reverberation decay time at low frequencies.
Decay HF ratio	High frequency decay time relative to low frequency decay time.
Reflections	Intensity level of early reflections relative to Room value.
Reflections delay	Delay time of the first reflection relative to the direct path.
Reverb	Intensity of late reverberation relative to Room value
Reverb delay	The time limit between the first reflection and the late reverberation.
Diffusion	Echo density in the late reverberation decay.
Density	Modal density in the late reverberation decay.
HF reference	Reference high frequency.

4.5 EAX

EAX stands for Environmental Audio eXtensions and is an extension either to Microsoft's DirectSound 3D that is a part of DirectX, or to OpenAL [OPENAL]. EAX 1.0 was released by Creative Labs in 1998 and it has been followed by EAX 2.0 in 1999 and EAX

Advanced HD in 2001 [Hagen, Muschett 2002] [Creative]. The properties of EAX 2.0 are described in this subchapter from the DirectX point of view. EAX is divided into the listener property set and the sound-source property set.

4.5.1 The listener property set

The listener property set gives extra controls for the primary buffer of DirectSound. However, in the newest versions of DirectX (versions 8 and 9), almost all the listener settings EAX 2.0 provides are already covered by I3DL2 reverb (Table 2). Moreover, the presets are the same with just a couple of exceptions.

One thing that EAX adds to DirectSound 9 is the possibility to set one higher-level property, namely apparent size of the surrounding "room". This environment size setting affects five lower-level listener properties: reflections, reflections delay, reverb, reverb delay, and decay time.

4.5.2 The sound-source property set

The sound-source property set gives additional settable properties for a secondary buffer. These properties are all new compared to existing settings in the DirectSound 9. The properties are listed in Table 3. They are the same as in I3DL2 except that Occlusion room ratio, air absorption factor, and outside volume HF are new.

Table 3. The sound-source properties of EAX 2.0. Properties marked with asterisk (*) do not exist in I3DL2.

Parameter	**Description**
Direct	Relative correction to the source's direct-path intensity.
Direct HF	Relative correction to the source's direct-path intensity at higher frequencies.
Room	Additive source-specific setting to the global listener room property.
Room HF	Additive source-specific setting to the global listener room HF property.
Obstruction	The amount of obstruction muffling affecting the source's direct-path sound at high frequencies.
Obstruction LF ratio	Obstruction attenuation at low frequencies relative to the attenuation at high frequencies.
Occlusion	The amount of obstruction muffling affecting both the source's direct-path and reflected sound at high frequencies.
Occlusion LF ratio	Occlusion attenuation at low frequencies relative to the attenuation at high frequencies.

Occlusion room ratio*	Additional amount of occlusion attenuation for reflected sound.
Room rolloff factor	Additive source-specific setting to the global listener room rolloff factor property.
Air absorption factor*	Multiplicative source-specific setting to the global listener Air absorption HF property.
Outside volume HF*	This property makes the source directivity pattern frequency-dependent setting separate high frequency attenuation to the rearwards radiation.

5. Java 2 Micro Edition

This chapter describes Java 2 Micro Edition that is the underlying platform of the novel API.

5.1 Different Java Platforms

In early 1996, Sun Microsystems Inc. released the first version of Java Development Kit (JDK), and since that many subsequent releases have followed. Initially, the Java language was targeted towards consumer devices such as interactive TV, but as time passed, the Java platform grew and it started to be targeted more towards desktop and enterprise computing. The consequence was that the large collection of libraries could not fit anymore to limited consumer devices. Sun realized the problem and grouped the new Java 2 Platform into three editions to better meet the requirements of different environments: Enterprise Edition, Standard Edition, and Micro Edition. Java 2 Platform, Enterprise Edition is targeted for enterprises in need of scalable server solutions, Standard Edition is for desktop computer market, and Micro Edition is for consumer and embedded devices. This thesis concentrates on Micro Edition (J2ME).

Highly optimized Java runtime environments of J2ME technology specifically address the large consumer space, which covers the range of extremely tiny products such as smart cards or pagers all the way up to the TV set-top boxes, devices almost as powerful as desktop computers. J2ME aims to maintain the qualities that Java technology has become known for: built-in consistency across products, portability of code, safe network delivery and upward scalability. J2ME allows device manufacturers to open up their devices for widespread third-party application development and dynamically downloaded content, without losing the security or the control of the underlying manufacturer-specific platform. Probably in the future, the majority of applications will be developed, instead of device manufacturers, by third-party developers.

While consumer devices have many things in common, they are also extremely diverse in form, function, and features. The devices can be operated in various ways: for instance, with keyboard, stylus, or voice. The range of existing device types and hardware configurations is large, and the technology is constantly improving rapidly. Also a diverse range of applications changes and grows in unforeseen ways all the time. To address this multidimensional diversity, an essential requirement for the J2ME architecture is not only small size but also modularity and customizability. This is supported by two essential concepts of J2ME environment: configurations and profiles. They are described in the following chapters. [Riggs et al 2001]

5.2 Configurations

A J2ME configuration defines a minimum platform for a “horizontal” category or grouping of devices, each with similar requirements on total memory budget and processing power. A configuration defines the Java language and virtual machine features and minimum class libraries that a device manufacturer or a content provider can expect to be available on all devices of the same category.

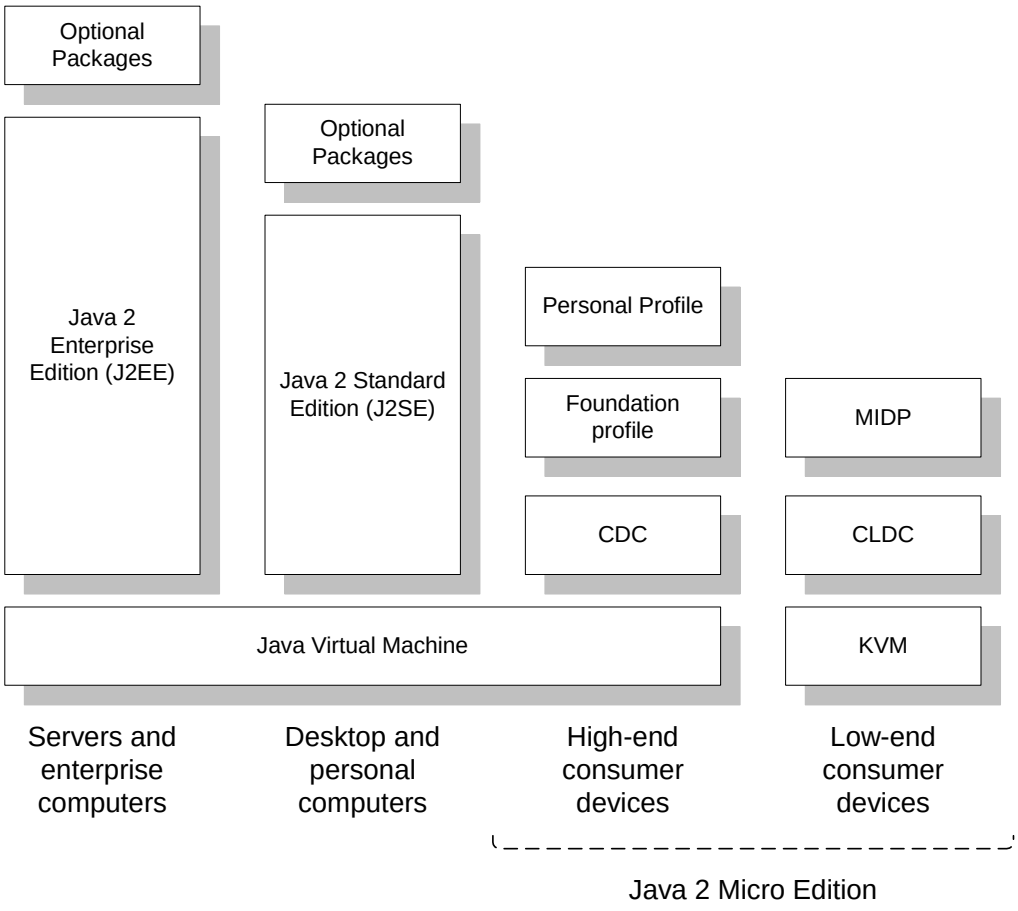

Figure 15. Java 2Platform editions.

To avoid fragmentation, there is a very limited number of J2ME configurations. Currently, only two standard J2ME configurations are available: Connected, Limited Device Configuration (CLDC) and Connected Device Configuration (CDC). They are visible on Figure 15 that illustrates Java 2 Platform editions. CLDC focuses on low-end consumer devices, that are typically personal, mobile, battery-operated, connected information devices such as cellular phones, pagers, and personal digital assistants (PDA). CDC’s focus area is high-end consumer devices, such as TV set-top boxes, Internet TVs, and high-end communicators. CDC includes a much more comprehensive set of Java libraries and virtual machine features than CLDC.

The majority of functionality in CLDC and CDC has been directly inherited from Java 2 Platform, Standard Edition. Each class inherited from the J2SE environment must be precisely the same or a subset of the corresponding class in the J2SE environment. In addition, CLDC and CDC can introduce features that are not drawn from the J2SE, but instead designed to specifically fulfill the needs of small-footprint devices. [Riggs et al 2001]

5.2.1 Connected, Limited Device Configuration

CLDC is the target configuration for the API defined in this thesis, so it will be described in more detail in this chapter.

Target devices of CLDC 1.1 are characterized in the specification [CLDC 2003] as follows:

- At least 192 kB of total memory budget available for the Java platform,
- a 16-bit or 32-bit processor,
- low power consumption, often operating with battery power,
- and connectivity to some kind of network, often with a wireless, intermittent connection and with limited bandwidth.

Typically, these target devices are manufactured in very large quantities, meaning that manufacturers are usually extremely cost-conscious and interested in keeping the per-unit costs as low as possible.

CLDC Specification addresses the following areas:

- Java language and virtual machine features
- Core Java libraries (java.lang.*, java.util.*)
- Input/output
- Security
- Networking
- Internationalization

Areas such as application life-cycle management, user interface, and event handling are not part of CLDC. They can be addressed by profiles implemented on top of the CLDC. Profiles are described in detail in a separate chapter later.

The virtual machine itself is a part of CLDC specification CLDC libraries being the other part. The virtual machine of CLDC is called as KVM[3] instead of the Java Virtual Machine (JVM) of J2SE. The goal for KVM was to be as compliant with the Java Virtual Machine Specification [Lindholm, Yellin 1999] as possible. However, with strict memory constraints of CLDC target devices, some features of J2SE JVM had to be left out from KVM. The most important of the dropped features that KVM doesn't have is Java Native Interface (JNI); the way in which the k virtual machine invokes native functionality is implementation-dependent. Other missing features include user-defined class loaders, reflections (and dependent features such as remote method invocation and object serialization), thread groups, daemon threads, and finalization. Also the set of Error classes included in CLDC is limited. CLDC 1.0 was lacking even floating point support [CLDC 2000], but it was added later in CLDC 1.1.

[3] The letter K comes from "kilo" to describe the unit of size that KVM is measured in: kilobytes.

Like the JVM of J2SE, the KVM has a capability to detect and reject invalid class files. However, since the standard class file verification process of J2SE requires quite a lot of memory and processing power, J2ME uses more compact and efficient verification solution. Instead of one heavy verification process done by the device, the task is now divided into two parts. There is a special pre-verification tool, which is typically used by the application programmer in the development environment. Then when running the application, the in-device verifier utilizes the information generated by the pre-verification tool to speed up the runtime verification process.

5.3 Profiles

A J2ME device profile is layered on top of a configuration. A profile addresses the specific demands of a certain "vertical" market segment or device family. The main goal of a profile is to guarantee interoperability within a certain vertical device family or domain by defining a standard Java platform for that market. Profiles typically include class libraries that are far more domain-specific than the class libraries provided in a configuration. [Sun 2000]

Profiles can be thought to be of two different categories: device-specific or application-specific. A device-specific profile serves as a common denominator for certain kind of Java enabled devices such as cell phones, washing machines, or electronic toys. An application developer can then write an application on top of a device-specific profile and that application will then work in all devices, say cell phones, which support that profile.

Another point of view is to create a profile for a specific application group. Then all the device manufacturers that want to support the group of that kind of applications are able to do that by implementing that application-specific profile in their devices. This is possible, because in J2ME, it is possible for a single device to support multiple profiles.

Profiles target specifically against the problem of portability in the J2ME domain that hosts such a wide range of devices. Grouping of devices is possible by categorizing devices by profiles they implement and application programmer can choose from limited amount of profiles which one to use without need to write device dependent code. New devices can take advantage of a large and familiar application base by just implementing an appropriate profile. Most importantly new applications can be dynamically downloaded to existing devices.

At the implementation level, a profile is defined simply as a collection of Java APIs and class libraries that reside on top of a specified configuration giving some domain-specific additional functionality.

5.3.1 Mobile Information Device Profile

Mobile Information Device Profile (MIDP) [MIDP 2002] is targeted for a well-defined group of devices, Mobile Information Devices (MID). A MID should have a display capable to display at least 96*54 resolution with black and white pixels. The aspect ratio of the pixels should be approximately 1:1. A MID should have one or more of the following user-input mechanisms: "one-handed keyboard" such as ITU-T phone pad, "two-handed keyboard" such as QWERTY, or a touch screen. The memory requirements for a MID are the following in the MIDP 2.0 specification:

- 256 kilobytes of non-volatile memory for the MIDP implementation

- 8 kilobytes of non-volatile memory for application-created persistent data
- 128 kilobytes of volatile memory for the Java runtime

Furthermore, a MID should have two-way wireless, possibly intermittent, limited bandwidth network. Additionally, a MID should be able to play tones. These requirements for a MID are typically fulfilled by modern cell phones and personal digital assistants (PDA) with network connections.

MIDs span a potentially wide set of capabilities, but MIDP addresses only those APIs that are considered absolute requirements to achieve broad portability. These APIs are:

- Application (i.e., defining the semantics of a MIDP application and how it is controlled)
- User interface (includes display and input)
- Persistent storage
- Networking
- Sounds
- Timers

The MIDP is designed to operate on top of CLDC and being so, the features of CLDC complement MIDP.

MIDP abandons the Applet model familiar from J2SE, because of strict memory limitations and requirement to support data sharing between applications. In MIDP, the basic unit of execution is a MIDlet. The MIDlet model is quite similar to the Applet model. MIDlet has one class that extends javax.microedition.midlet.MIDlet and possibly other classes that are needed by MIDlet. These classes are packed in a JAR file with standard JAR Manifest and optional other files such as pictures or sound files. Each JAR file may be accompanied by an application descriptor. The application descriptor allows the application management software on the device to verify that the MIDlet is suited to the device before loading the full JAR file of the MIDlet suite. It also allows configuration-specific attributes (parameters) to be supplied to the MIDlets without modifying the JAR file.

Like application model (MIDlet), MIDP also introduces a new user interface (UI) model. Abstract Windowing Toolkit (AWT) of J2SE is optimized for desktop computers and is not suitable for MIDs. AWT was abandoned mainly because of the following reasons: it is based on windows and MIDs are not in general capable of displaying overlapping windows; AWT was designed to work with a pointer device unlike most of MIDs; and furthermore, AWT uses dynamically generated event objects that might be overwhelming to MIDs limited CPU and memory capacity.

LCDUI, the new UI model of MIDP is logically composed of two APIs: the high-level and the low-level. High-level API employs a high level of abstraction and offers a complete user interface components such as alerts, lists, text boxes, and forms. The actual drawing to the MID's display is performed by the implementation and the visual appearance is not definable by the application programmer. Furthermore, navigation,

scrolling, and individual user input keys are not accessible directly by the application. This makes the applications to maintain uniform look and feel of the hosting device.

The other UI API, the low-level API offers very little abstraction. This API is designed for applications that need precise placement and control of graphic elements, as well as access to low-level input events. Some applications also need to access special, device-specific features. Applications that use the low-level API are not guaranteed to be portable, since the low-level API provides the means to access details that are specific to a particular device. Being so, programmers using the low-level UI API should take care of portability when designing applications by among other things inquiring on the size of the display and then adjusting their drawing routines accordingly.

5.4 Mobile Media API

There was no generic sound or multimedia support directly in CLDC 1.0 or even in MIDP 1.0 [MIDP 2000]. This generated a need to produce an extension to provide multimedia support for J2ME environments. Mobile Media API 1.0 [MMAPI 2002] was the resulting specification. It can be implemented, for instance, together with MIDP 1.0 (Figure 16) and actually, the new MIDP 2.0 already contains a limited audio-only subset of MMAPI. The standardization was done in Java Community Process (JCP) during years 2001 and 2002. JCP is described later. So far at least mobile phone models Nokia 3650, Nokia N-Gage, and Sony Ericsson T610 support MMAPI.

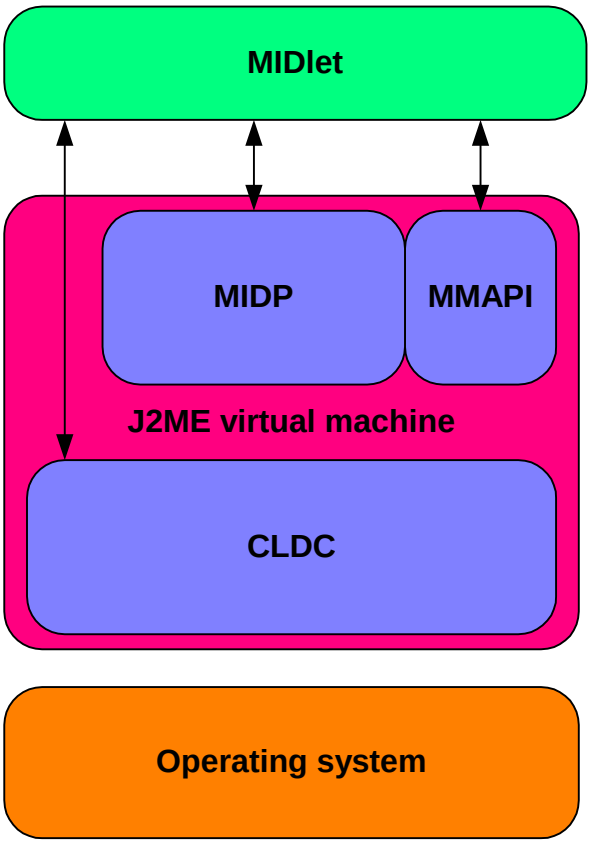

Figure 16. MMAPI can be implemented, for instance, to coexist with MIDP 1.0 to support multimedia needs of a MIDlet.

5.4.1 Features

The main feature of the Mobile Media API (MMAPI) is to offer tools for playback of any time-based media content, such as audio, video, or animation. The API has a high level of abstraction: it is generic what comes to protocol used to transfer the media or content type used to store the media. This addresses to the needs of the modern world where media types and formats keep evolving all the time just like their storage devices and transmission paths and protocols.

In addition for supporting any external media types, MMAPI also has a media type of its own. In the lowest end of mobile devices, simple buzzer tones might be the only multimedia capability supported. Keeping this in mind, MMAPI offers internal format for simple tone sequences. MMAPI's tone sequence consists of byte array with tone-duration pairs to form melodies and additionally support for defining blocks of tones to enable easy repetition of song parts.

MMAPI offers multiple ways to control how the media content is played back and handled. Not all the target devices that use MMAPI have capabilities to support such wide variety of different ways to control media. The design of MMAPI allows some of the features been left unimplemented and thus MMAPI support is possible also for devices with not so rich set of features. On the other hand, some devices might be capable of processing media in some ways that MMAPI does not have support. This problem is solved with MMAPI's extensible structure; it is possible to add new features without breaking old functionality and being compatible to original MMAPI.

CLDC is the main target of MMAPI and this sets strict memory consumption limitations. MMAPI is designed to have as small footprint as possible and this means that the API does not offer any unnecessary parallel, overloaded methods for the same functionality.

5.4.2 Structure

MMAPI divides multimedia processing into two separate conceptual parts: into protocol handling and content handling. Protocol handling takes care of the reading of the data from the source and transmitting it to media processing system. The source can be anything from the network server to some capturing device like microphone or camera. After protocol handling, content handling processes the data and renders it to output devices. Processing can mean, for instance, parsing and decoding the multimedia format.

MMAPI has two high-level objects to represent the dual nature of multimedia processing: DataSource and Player. The former takes care of the protocol handling and the latter of the content handling. A DataSource reads data from a source and then offers methods for a Player to read data from the DataSource itself, hiding the details how data is actually obtained from the source.

A Player reads data from the DataSource, processes data, and renders finally it to the output device. Player has methods for starting and stopping the media playback. Player has also a multistage life-cycle model. The purpose of different life-cycle stages is to give a possibility to limit the usage of scarce multimedia resources to as small as possible. Stages are in the order of usual occurrence in straightforward onetime playback the following: UNREALIZED, REALIZED, PREFETCHED, STARTED and CLOSED. The state transitions are illustrated in Figure 17.

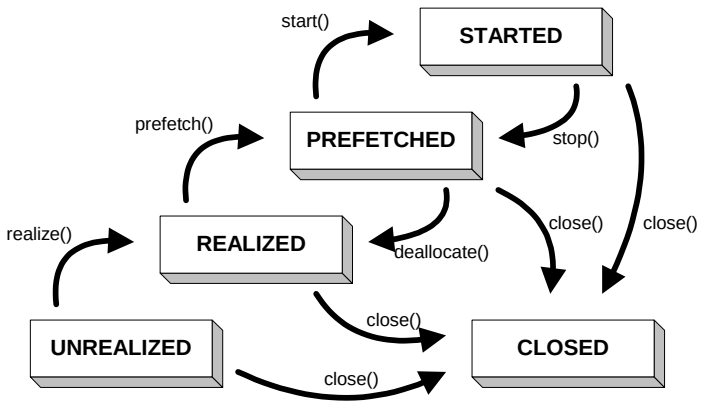

Figure 17. State transition diagram of the Player.

A Player is unrealized when it has just been instantiated. Realized Player has typically opened the media stream and prefetched Player has furthermore opened the scarce media resources of the device. Started Player is playing the media and closed Player has been closed and cannot been used again. Player offers methods for controlling transitions between the stages; also falling back in the stage hierarchy and thus freeing a resource when necessary is possible by these methods.

Players are instantiated with static factory methods of a class called Manager. Manager acts as a common access point for obtaining these system dependent resources. Manager ties DataSource and Player together and the factory methods return specific Players for wanted media type.

Furthermore, a Player acts as a factory for media type specific controls. These controls implement the interface Control and offer possibilities to control media properties such as volume, pitch, tempo, and metadata. Closer look into various Controls is taken in the next sub-chapter.

5.4.3 Controls

MMAPI offers the following pre-defined Controls: FramePositioningControl, GUIControl, and VideoControl for controlling media that is in visual form; VolumeControl, PitchControl, MIDIControl, TempoControl, and ToneControl for audio data; MetaDataControl for metadata; and finally StopTimeControl and RateControl to control playback flow in general level. Furthermore, there is RecordControl for recording purposes.

VolumeControl offers methods for setting and getting the sound level in linear point scale with values from 0 to 100. Also muting is supported.

With PitchControl it is possible to shift the pitch of the played back audio. No absolute pitch can be set; only relative values are supported. Pitch shift is expressed in millisemitones compared to original pitch of the media.

RateControl can be used to change the speed of the media playback. Rate is expressed in millipercents of the original rate. By setting negative value, it is possible to play the media in reverse. Also TempoControl affects the playback speed, but it is independent of RateControl and uses absolute values expressed in thousandths of beats-per-minute (milli-BPM). Furthermore, tempo set by TempoControl is volatile and changes e.g. when played MIDI sequence contains META tempo events unlike RateControl's rate.

MIDIControl gives advanced, low level functionality to control MIDI playback, but VolumeControl, TempoControl, and PitchControl are sufficient for most needs what comes to basic MIDI playback.

ToneControl acts as an interface for MMAPI's internal tone sequence format. ToneControl has fields defining the semantics of the format and then a method for taking the sequence into use for that Player.

RecordControl offers methods for setting the output locator for the recorded media and various controls for the recording itself. The media currently played by the Player will be recorded.

If media is wanted to be played just until some point in its internal media time, with StopTimeControl that time of stopping can be defined.

If there is a need for the Player to supply components for the graphical user interface (GUI), they can be provided by GUIControl. It has a functionality to create appropriate GUI primitive of the platform. For example, for MIDP platform that will be an object extending javax.microedition.lcdui.Item. VideoControl is a special case of GUIControl for displaying video on the GUI. VideoControl has methods for accessing displayed video's size and location. It also provides snapshots of the displayed video if required. With FramePositioningControl, it is possible to seek displayed video by frame-by-frame basis. In some implementations so accurate seeking might not be possible, but the best effort will anyway be given.

Metadata means data about data. This data can be accessed by MetaDataControl in MMAPI. Metadata of the media is identified with keys. Four pre-defined keys are available as fields of MetaDataControl: keys for copyright information, author, title, and date.

5.5 Java Community Process

Like CLDC and MIDP, Mobile Media API was also specified using Java Community Process (JCP) [JCP 2001]. JCP is an open organization that develops Java technology specifications, reference implementations, and technology compatibility kits. JCP is a formal process that makes the Java platform evolve. The JCP has over 300 company and individual participants.

The new technology specifications for Java that JCP produces are called as Java Specification Requests (JSR). JSRs are the actual descriptions of proposed and final specifications for the Java platform. JSR development work is carried out in so called Expert Groups. They usually consist of many companies of the field from all over the world. An Expert Group should be large enough to guarantee reasonable industry representation to later support the new specification. For every JSR a new Expert Group is formed and multiple JSRs are on the process simultaneously. The development of a new JSR is done on multiple steps where improved versions of the specification are produced and then reviewed and accepted by voting.

JSR-30, JSR-139, JSR-37, JSR-118, and JSR-135 specify CLDC 1.0, CLDC 1.1, MIDP 1.0, MIDP 2.0, and Mobile Media API, respectively.

6. Advanced audio API[4]

This chapter describes the novel API designed in this thesis. First the major requirements of the API are gone through, then the API itself is described, and finally the reference implementation done is presented.

6.1 Goals

6.1.1 Targeted users

The targeted user group of this API was set to be application programmers that want to add some advanced audio effects to their applications running in mobile devices. It had to be taken into consideration that the programmers do not necessarily have special knowledge of audio processing or psychoacoustics. The API had to offer enough presets to make it usable for non-audio experts.

A typical user was thought to be a game programmer that wants to enhance sensation of an acoustical environment with some characters moving around in space. He or she might also want to dynamically filter sounds with equalizer or some effect to simulate some change in a sound source.

6.1.2 Targeted platforms

The target environment for the API was chosen to be CLDC devices and this requires the API to be compatible with J2ME CLDC. API was also supposed to be compatible with Mobile Media API 1.0 so that it is a natural extension to it. However, a couple of small additions had to be done to original MMAPI 1.0. The size of API code should be as small as possible to keep the footprint of the virtual machine (KVM) small.

In the following chapters, the features that aim to fulfill the needs of the target users and platform are described. These features form the set of requirements.

6.2 Features

6.2.1 General

The API had to be easy and intuitive to use also for programmers that do not have any special audio knowledge. Most of the audio features have preset modes to make it possible for the user to easily adjust the parameters without having any special audio knowledge of

[4] The Advanced audio API and the associated documentation presented herein are copyright © 2003 Nokia Corporation. All rights reserved. No part of the Advanced audio API or its documentation may be reproduced in any form by any means without prior written authorization of Nokia.

what would be the appropriate values of the parameters. Special care was taken when designing the names of these preset modes. The names of the preset modes had to be descriptive, but compact. The Javadoc documentation was intended to be the self-contained, complete programming documentation for the API.

6.2.2 Source localization

One of the main features of this API is the possibility to locate sound sources in a virtual space. Human can perceive direction of a sound source quite accurately using certain physical properties of the sound arriving to ears. These so called location cues can be simulated by a computer and thus, a virtual direction for arriving sound can be created.

The distance has an effect of attenuating the far positioned source and also lowering the relative level of the source compared to the reverberation level. This being the case, giving just the direction of the source is not enough, because the distance of the source can also be simulated.

This API uses an approach where the sound sources have a location primarily defined in 3D-coordiantes so that both the distance and the direction are specified by just giving the XYZ-coordinates. 3D-coordination and its representation are compatible with Mobile 3D Graphics API for J2ME [M3D 2003].

API supports changing the location of a sound source smoothly. It is possible to give two locations for the source and force the source to move from one location to the other. The speed of sound sources moving in space relatively to the listener is definable. This could be done, for example, by defining the period of time when the movement is going to be completed.

6.2.3 Reverb

Reverberation is essential in perceiving the properties of the room. Mainly it tells the size of the room and gives some hints about the wall materials.

The API was chosen to offer several preset modes available for reverberation. A subset of the modes from “IA-SIG Interactive 3D Audio Rendering Guidelines (Level 2)” [I3DL2 1999] were used, but the goal was not to entirely fulfill the details of the reverberation model presented in it. Mobile devices with limited processing capacity might not always be capable of fulfilling I3DL2 completely.

The preset modes were to be tunable with a single simple parameter, reverberation time. It makes the preset modes more flexible. The presets were chosen so that they span through wide range of different reverbs and with the capability of adjusting the reverberation time, many different kinds of reverberations came available.

Because reverberation algorithms are computationally really laborious, it was decided that one common reverb is enough for all the sound sources; no individual reverberation parameters can be set to sources. Although the reverb is common, bypassing single sources is possible; this makes it possible to use sources that are non-anechoic recordings and already reverberant.

6.2.4 Equalizer

Equalizers are usually used for two reasons: to compensate unideal frequency response of a system to make it sound more natural or to create intentionally some unnatural coloring

to the sound to create an effect. The equalizer in this API has to be able to serve both of these purposes.

Like the reverb, the equalizer has also several preset settings available. Modes like telephone band, bass boost, loudness, karaoke, subsonic cutoff, ultrasonic cutoff, and muffling could be available for easy using. Also some programmers might want to use the traditional EQ presets named after musical genres (Classical, Jazz, Rock, Pop).

There are also convenient methods for altering bass or treble only without need to specify any frequencies. Most of the people are familiar with these traditional settings of consumer audio devices.

There are methods for getting and setting individual EQ-band gains and a method for asking the amount of the EQ-bands available. People that are more familiar with EQ are able to create effects or to compensate the response of the system with these methods.

6.2.5 Effects

The API was chosen to support arbitrary audio effects via a general interface. A user can ask for the available effects in the device and then choose which ones to use. Here we mean by effects the common ways to process sound of some musical instrument. Most of the musical effects originate from electrical guitar footswitches, but the effects are not limited to be used just with guitars.

The most important parameters of the effects had to be made controllable. For instance, chorus, flanger, and phaser effects have at least the following common parameters: blend, frequency of low frequency oscillator (LFO), and amplitude of LFO. Different effects might also have different non-common parameters that must be supported, too.

The effects can be inserted in two places on the signal path. They can be inserted as source specific effects or as global effects. The computationally easiest effects should be channel specific so that they can be tuned for each sound source independently. The computationally most demanding effects can be tuned only globally for all the sources together.

6.3 Process

The interface was defined using an iterative process. The initial version was written by the author and then reviewed by the colleagues. Altogether three reviews were organized and the interface was always tuned based on the feedback got in the previous review session.

The API was first written using empty Java methods with thorough Javadoc commenting and then Javadoc tool was used to generate the documentation in the HTML form.

6.4 Interface

Advanced audio API (AAI) consists of Java interfaces that supplement the classes and interfaces of MMAPI. AAI provides a new interface called as Spectator that is used to control directly or via other classes the listener model and the room model of the virtual acoustical space. New Controls are introduced as well: EQControl, LocationControl, OrientationControl, PanControl, and EffectControl. EffectControl utilizes new interface

Effect and its sub-interfaces Reverb and Chorus. All these interfaces are described in detail in the following subchapters.

6.4.1 Spectator

Spectator is a representation of a human spectator of Players in virtual acoustical space. Spectator is used to control directly or via other classes the listener model and the room model of virtual acoustical space.

Spectator implements Controllable interface, and can thus create at least five different Controls for itself if they are supported by underlying audio engine: VolumeControl, EQControl, EffectControl, LocationControl, and OrientationControl. They are used to configure global volume, global equalizer, global effects, and Spectator's location and orientation respectively. Some global properties such as volume and equalizer might also be controllable directly from the device's native user interface. Spectator's relations to the other interfaces are illustrated in Figure 18.

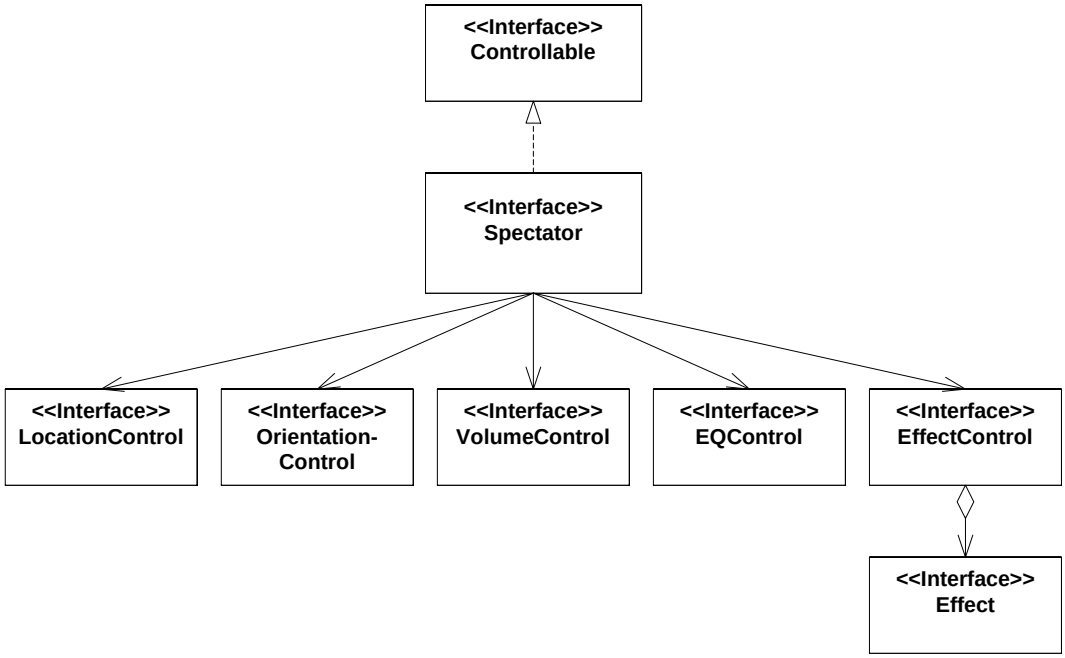

Figure 18. UML class diagram about Spectator's relations to the other interfaces.

A Spectator can be instantiated with Manager.createSpectator() in a similar way as Players are instantiated with Manager.createPlayer(). There is typically just one instance of a Spectator in an application. Anyway, multiple instances can be created for multiple simultaneous human users of the application if the underlying audio engine supports that and there are multiple output devices available. Manager.createSpectator() returns a Spectator that is located in the origin and points directly towards the negative z-axis direction.

A Spectator has two life-cycle states: ACTIVE and CLOSED. The purpose of these life-cycle states is to provide programmatic control over potentially time-consuming operations. When all the Players are gone, the Spectator is not automatically closed. This makes it possible to maintain settings of Spectator, such as the global Effects, through multiple Player life cycles. On the other hand, sometimes it is necessary to release all the resources of a Spectator and not use them again. This is possible with close() method.

When a Spectator is first constructed, it is in the ACTIVE state. An ACTIVE Spectator means that the Spectator is running and processing data. Calling close on the Spectator puts it in the CLOSED state. In the CLOSED state, the Spectator has released most of its resources and cannot be used again. All the settings made with Spectator's Controls are then lost.

The location and the orientation of a Spectator can be specified with the Controls LocationControl and OrientationControl Spectator offers.

6.4.2 LocationControl and OrientationControl

LocationControl is an interface for manipulating the virtual location of an object (usually a Player or a Spectator) in virtual acoustical space.

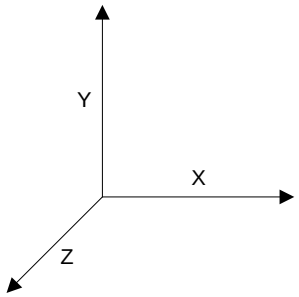

Figure 19. Location is defined in right-handed coordinates.

Location is a representation of a place in a virtual acoustical space. This interface allows the location to be specified in three-dimensional space with method setLocation(). The method has possibility to set the new location immediately (by giving zero as parameter) or to set it slowly so that the movement is exactly completed after given time.

The initial location of the Player has to be set before prefetching the Player, because it also takes care of the initialization of the spatialization. If the location is set first time when the Player has been already prefetched, the spatialization is not necessarily active.

OrientationControl is an interface for manipulating the virtual orientation of an object (usually Spectator) in virtual acoustical space.

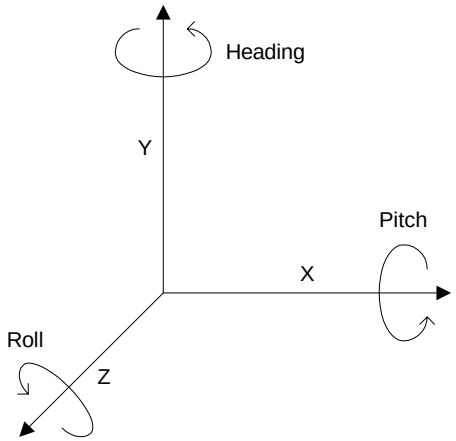

Figure 20. Orientation is defined in right-handed coordinate system.

Orientation is a representation of a direction in a virtual acoustical space. Orientation is defined in rotation angles around coordinate axes. Positive rotation directions around the coordinate axes are counterclockwise when looking towards the origin from a positive coordinate position on each axis. The initial orientation is towards negative Z-axis. Orientation is defined in three angles: heading that represents rotation around Y-axis, pitch that represents rotation around X-axis, and Orientation that represents rotation around Z-axis. Orientation is applied in the following order: heading, pitch, and roll.

6.4.3 PanControl

PanControl is an interface for manipulating the panning of a Player in the stereo output mix.

This interface allows the panning between the left and right channels to be specified using an integer value that varies between –100 and 100. The pan scale specifies panning in a linear scale. It ranges from –100 to 100, where 0 represents panning for both channels, -100 full panning to the left, and 100 full panning to the right. The mapping for producing linear multiplicative values is implementation dependent. With stereo sources, the effect of the panning set is undefined.

API supports changing the panning of a sound source smoothly. The setPan method has possibility to set the new panning immediately (by giving zero as duration parameter) or to set it slowly so that the change is exactly completed after given time. The panning setting is not in effect if the location of the source has been specified with LocationControl and the spatialization is in use.

6.4.4 EQControl

EQControl is an interface for manipulating the equalization settings of a Player or a Spectator.

This interface allows the sound source to be filtered with an equalizer. The equalizer can be set up with three different ways: using presets, using bass and treble controls, or by setting individual frequency bands by hand.

The preset settings can directly be taken into use with method setPreset(). The current preset can be found out with method getPreset().

There are convenient methods setBass() and setTreble() for altering bass or treble only without need to specify any frequencies. SetBass() and setTreble() reset the current set up on the band on question (bass or treble frequencies respectively). There are also methods for asking current bass and treble setting (getBass() and getTreble()), but if the equalizer settings have been altered after bass and treble set-up by setting presets or altering individual EQ-bands, bass and treble settings are not anymore unambiguously defined and EQControl.UNDEFINED will be returned.

There are also methods for getting and setting individual EQ-band gains (setLevel() and getLevel()) and methods for asking the amount of the EQ-bands available (getNroOfBands()) or their center frequencies (getCenterFreq()). People that are more familiar with EQ are able to create effects or to compensate the response of the system with these methods.

The gains in this class are defined in decibels, but it has to be understood that many MIDs contain a dynamic range control (DRC) system that will affect the actual effect and therefore, the value in decibels will affect as a guideline rather than as a strict rule.

6.4.5 Effect, EffectControl, and PlayerEffectControl

Effect is an abstract audio filter with various preset settings. Individual Effects might have various parameters. All Effects have accessors for effect level. The effect level affects how much of the sound is passed via the effect in percents. 100 means that all the sound is processed in the effect and 0 means that no processing is done at all and all the sound is bypassed. Values between 0 and 100 affect the wet/dry ratio of the processing in the accuracy that the system supports. The effect is considered to be "wetter" when the effect level rises. Effects are presettable like EQ and can be instantiated with createEffect() of the EffectControl.

EffectControl controls the effect box behavior of a Spectator or a Player. An effect box is a combination of various Effects. It is like a multieffect whose individual effects can be added or removed. Spectator and Player can have EffectControls. Being so, they can use zero or more Effects.

With certain kind of effects (non-linear effects), the processing order affects the result. It is possible with this interface to specify the wanted processing order by inserting effects with an index number (method insertEffect()) to the signal path, but the proposed order is not necessarily obeyed; this depends on underlying audio engine's capabilities. When inserting effects in the middle of the effect chain the previous effect on the insertion place and all the subsequent effects will be shifted one step to make space for the new effect. Usually, in the case of global Effects (the Effects of Spectator), the order of the Effects doesn't count, because the Effects are parallel on the signal path, unlike typical serial processing order of the Player-specific Effects.

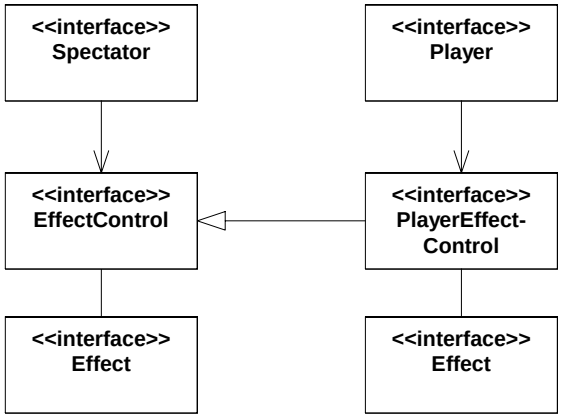

Figure 21. UML class diagram about EffectControl and associated interfaces.

PlayerEffectControl extends the interface EffectControl to include methods for setting effect send levels. PlayerEffectControl controls the effect box behavior of a Player.

The important property of a PlayerEffectControl is that it can be used to control the level of effects in other affecting effect boxes as well. For example, if there is a Player called p1 that one Spectator is listening among other Players, and that Spectator has a global effect, say Reverb, turned on, p1 can be bypassed from this global Reverb by using a method call

((PlayerEffectControl)(p1.getControl(“PlayerEffectControl”))).setGlobalEffectSendLevel(0);. Other Players will continue using the global Reverb normally.

6.4.6 Reverb

Reverb is an interface for manipulating the settings of an effect called reverb. Therefore, Reverb implements Effect interface. A Reverb can be instantiated using method EffectControl.createEffect(“Reverb”).

Reverberation is essential in perceiving the properties of the room. Mainly it tells the size of the room and gives some hints about the wall materials.

In the API, there are several preset modes available for reverberation. A subset of the modes from “IA-SIG Interactive 3D Audio Rendering Guidelines (Level 2)” [I3DL2 1999] is used. The modes at least available for the reverb are:

- alley
- arena
- auditorium
- bathroom
- cave
- hallway
- hangar
- livingroom
- mountains
- room

The preset modes are tunable with a single parameter, the reverberation time. It is a single intuitive value that will affect how reverberant the acoustical space sounds. This parameter has a default value for every preset reverberation mode and then the user can, for example, make the preset room more reverberant by raising the value. Methods for setting and getting the reverberation time are setReverbTime() and getReverbTime(). It is recommended to first get the preset’s reverberation time, then scale it, and finally set the new, scaled value.

6.4.7 Chorus

Chorus is an interface for manipulating the settings of an effect called chorus and its special case flanger. Chorus, like Reverb, implements Effect. A Chorus can be instantiated using method EffectControl.createEffect(“Chorus”).

Chorus makes the sound source sound like a group of similar sound sources playing the same sound. This makes the sound “richer”.

A special case of a chorus is a flanger: two similar sources are played so close together in time that they sound like one, but instead with “wooshing” effect or with a sound similar to the sound of a jet plane passing overhead.

In the API, there are several preset modes available for chorus. The modes at least available for the chorus are:

- chorus
- flanger

Minimum tunable parameters for the chorus (and the flanger) are average delay and delay modulation’s rate and depth.

6.5 Comparison to other 3D audio APIs

This section compares the features of the presented APIs, namely Java 3D 1.3, MPEG-4, DirectX 9.0, EAX 2.0, A3D 3.0, and AAI.

6.5.1 Geometry

There are basically two different ways to define the geometry in the virtual acoustical world: scene graph based that describes the world in a treelike structure and non scene graph based that does not use any particular structure and instead describes the sound source and listener locations and possible walls essentially as a list.

Of the presented APIs, Java 3D and MPEG-4/BIFS are scene graph based and on the other hand, DirectX, A3D and AAI are list based. EAX cannot be classified because it does not define geometry being an extension to DirectX that defines the geometry.

BIFS and A3D have the possibility to introduce walls that have acoustical properties into the world in addition to sources and listeners that other APIs only provide. BIFS uses scene graph for walls and A3D offers special lists for wall vertices that can be defined once but used multiple times.

6.5.2 Room effect

Different APIs have different settings for the reverberation. DirectX, EAX and A3D have settings similar to, or the same as, I3DL2. DirectX has exactly the same parameters and presets that I3DL2 defines: 12 parameters (Table 2) and 30 presets. EAX and A3D have only slightly different presets the amount being 26. The parameters are the same as in I3DL2 except that EAX provides an additional high-level “room size” parameter. AAI’s reverb is also I3DL2 based offering a 10-preset subset that can be controlled with one parameter, reverb time (T60).

Java 3D offers an eight-parameter subset of I3DL2 parameters, but does not have any presets. Instead of presets, Java 3D offers an easy way to calculate reverb decay and delay automatically internally: the application can define the boundaries of the room and then reverb decay and delay are calculated accordingly.

BIFS has two alternative ways to set the reverb: perceptual and physical. The perceptual approach offers 13 parameters to control reverberation. They are of really perceptual

nature, like source brilliance and room presence. The other approach is physical where different acoustical materials are bound with the geometry defined for the room (walls).

Since BIFS and A3D are the only APIs that have capability for wall geometry definitions with wall material, they are the only APIs that support automatic calculation of the early reflections, obstruction and occlusion. However, EAX offers an option to manually set magnitudes for the obstruction and occlusion for each source, but that requires quite a lot manual calculations of walls' shadowing. Similar manual obstruction and occlusion settings are possible also with AAI's EQControl.

6.5.3 Source directivity

Java 3D and BIFS have fine-grained control for source directivity: multiple angles from the source main axis are defined and then the gains and the frequencies for the low-pass filters are defined for those angles.

DirectX and A3D have only two definable angles and the remaining directivity is then interpolated between them. The gain outside of the outer angle is definable in both APIs, but it is frequency independent. EAX adds frequency dependent directivity to the DirectX.

In addition to directivity, A3D has possibility for volumetric sources.

AAI does not provide automatic directivity calculations, but frequency dependent source directivity can be manually implemented using EQControl.

6.5.4 Effects

Sometimes it is required to be able to alter sound also in unnatural ways, in contrast to the natural environment modeling such as 3D localization and reverberation. From the set of presented APIs only AAI and DirectX have predefined effects (apart from reverb) available. AAI provides three: chorus, flanger, and graphical equalizer; and DirectX, on the other hand, has eight standard effects: chorus, compression, distortion, echo, flange, gargle, parametric equalizer, and Waves reverberation. Also other user-specified effects can be registered into the both systems.

Although MPEG-4/BIFS does not provide predefined effects, it has the most fundamental possibilities to define effects. AudioFX nodes can be added into the signal processing tree and SAOL is used to define them. In other words, a specialized programming language is provided for effects programming.

Java 3D and A3D do not provide direct help for the effect creation, but manual processing of the input streams is always possible.

AAI and MPEG-4 allow the effects to be either global, thus affecting all the sources, or source-specific. The rest of the APIs allow source-specific effects only.

6.5.5 Resource consumption control

Processing of sound data is in general a laborious task. The APIs offer different ways to try to deal with that.

Deferred settings is one way. It means that while an application sets processing parameters, such as source locations, they are not taken into action right away, but instead accumulated into a buffer. The settings in the buffer can then be processed together when

a sufficient amount of settings have been accumulated. DirectX and EAX have deferred settings and also A3D has a similar system, but it is called instead as a frame buffer.

A3D has a system called as a resource manager that has an ability to control, in a clever way, which of the sources fall back on software processing in the lack of hardware resources and further on no processing at all in the lack of software processing resources. The audibility and the priority of the source affect this fallback process.

The resource consumption of the reverb can be limited in Java3D, in A3D and in DirectX. DirectX also provides a choice to use two alternative HRTFs for 3D calculations: “light” or “full”.

MPEG-4 and AAI do not provide any ways for process consumption optimization. In the case of AAI, it is clearly a weakness; AAI will mostly be run on the devices with limited processing power.

6.6 Reference implementation

As a part of this thesis, a reference implementation of Advanced audio API was implemented. The reference implementation runs on top of Microsoft’s Windows 2000 operating system and provides the implementation of AAI’s LocationControl, OrientationControl, PanControl, EffectControl, Effect, Reverb, and Spectator.

The reference implementation of AAI is constructed on top of the MMAPI 1.0 Windows reference implementation and thus is based on MIDP 1.0.3 and CLDC 1.0.

The audio processing is done by a native audio engine running on the Windows platform. This engine is not part of the thesis.

6.6.1 Playback architecture

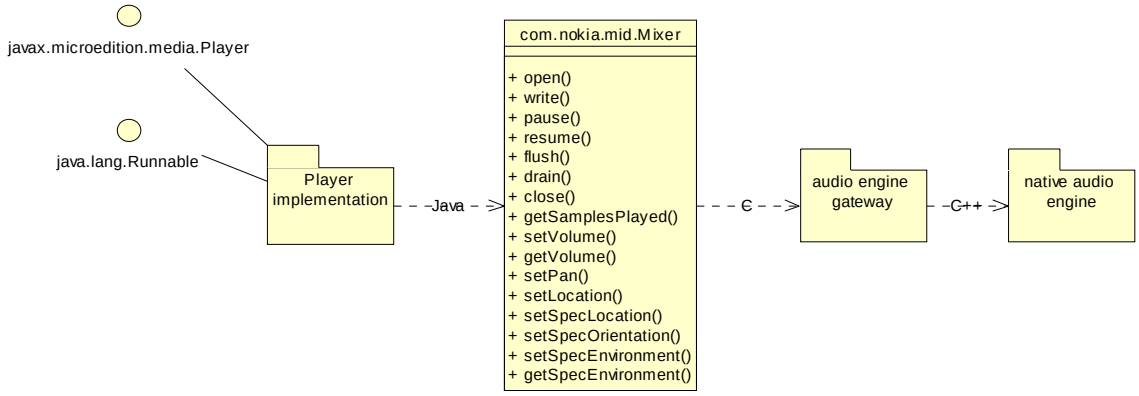

Figure 22. The architecture from the audio playback point of view. Only the relevant methods are shown.

Figure 22 describes the basic architecture that enables the playback of a wav-file. The original Player implementation creates a separate Thread that has the functionality to repeatedly read a bufferful of data from its input stream and then send it to the Mixer’s write method that sends it further to the native audio engine. In the original MMAPI reference implementation, Mixer class was replaced by an interface for Windows’ native audio resources.

One complication here is that CLDC's native interface supports calling C functions only [KVM 2001], but the native audio engine uses C++ classes as its public interface. Therefore, a gateway is needed to convert the calls. As C is a non-object language (unlike Java or C++), all the object behavior in the traffic between them has to be emulated by the Mixer and the gateway. An extra integer argument, namely handle, is passed in every C function call to represent the object that cannot be directly passed via C.

6.6.2 Controlling the playback

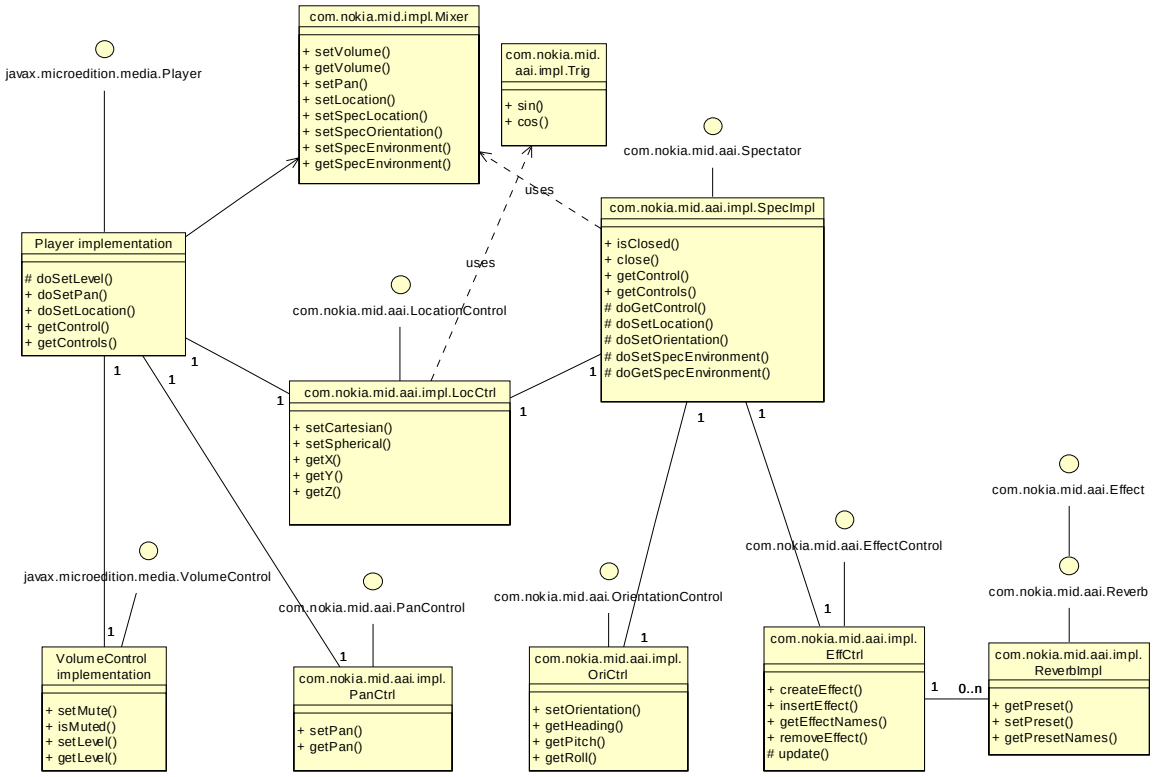

Figure 23. The architecture from the control point of view. Only the relevant methods are shown.

Figure 23 illustrates how the different Controls are implemented and also how a Spectator is added parallel to Player.

SpecImpl class implements the Spectator functionality. It uses Mixer, likewise Player, to control the native audio engine. Parameters that a Spectator controls are all global so no instance of Mixer is needed to be associated; only calling the Spectator related static methods of Mixer is enough.

Controls supplementing a Player are the original VolumeControl implementation together with author's PanCtrl implementing PanControl, and LocCtrl implementing LocationControl. Spectator's Control implementations by the author are OriCtrl implementing OrientationControl, EffCtrl implementing EffectControl, and the same LocCtrl that can also control Player.

All the Controls work so that when they are asked to modify some associated audio processing parameter they call the associated method of their "host". These associated methods are named with a prefix "doSet". For instance, when the application programmer wants to turn the Spectator in the virtual world, he or she calls Spectator's

OrientationControl's setOrientation method and gives the new orientation. In practice, OriCtrl.setOrientation is called and it passes the new orientation to SpecImpl.doSetOrientation that furthermore passes the new orientation angles to Mixer.setSpecOrientation.

LocCtrl is the only Control that needs direct native support in this API. LocCtrl needs trigonometric functions to convert the coordinates given in spherical coordinate system to the Cartesian coordinate system. Unfortunately, neither MIDP 1.0 nor CLDC 1.0 provides trigonometric functions, so a helper class Trig was build. Trig gives access to the native sine and cosine implementations.

EffCtrl that implements EffectControl functionality provides programmer a factory method, namely createEffect, to instantiate various Effects. The only effect implementation available here is ReverbImpl that implements the Reverb interface. ReverbImpl provides all 30 presets that "IA-SIG Interactive 3D Audio Rendering Guidelines (Level 2)" [I3DL2 1999] describes, not just the subset that the AAI requires. Reverb settings work in the way that when the application programmer calls the ReverbImpl.setPreset the EffCtrl's protected method update is called. EffCtrl.update reads the associated ReverbImpl's settings and then passes them to SpecImpl.doSetSpecEnvironment that passes them further to the Mixer. EffCtrl.update checks that the Reverb is inserted into the EffectControl before sending any parameters further.

The whole architecture of the reference implementation was redesigned many times during the work to make it iteratively compact, simple and efficient. For each iteration, some methods or even whole classes were dropped and the implementation became more compact.

7. A demonstration application

A demonstration MIDP application was build both for demonstrating and testing the novel API. In this chapter, the features of the demonstration MIDlet are first described and then the implementation is gone through.

7.1 Description

The demonstration MIDlet consists of a house where the user can walk around. He or she can walk through corridors and enter different rooms. Besides the user, there are other creatures in the house. They also move around the house, but their movement cannot be controlled: it is random.

Different creatures make different sounds. There are, for instance, dogs, different kind of birds, and some musicians performing opera. The creatures act as point sound sources in the virtual acoustical world and the user hears their relative direction. The relative directions change constantly while the sources and the user move in the virtual world.

Different rooms in the house have different kind of acoustical conditions. The reverb setting in use is dependent on the room where the user in the particular moment is located in. The locations of the sound sources are monitored constantly and compared with the room coordinates; those sources that are in the other rooms behind a door are completely occluded.

In addition to the virtual acoustical world, there is a graphical user interface where all the sources and the user are visible in the floor plan. The floor plan turns and scrolls while the user moves. So the user is always visible constantly on the center of the screen and facing upward. The user has five actions that he can do: move a step forward or backward, turn ten degrees left or right, or, by pressing “fire” button, change actively the reverb setting of the room where he is located at the moment. The display shows the coordinates of the user in millimeters, user’s rotation in degrees, and the reverberation preset of the room. The graphical user interface is visible in Figure 24.

Figure 24. A screenshot of the AAI reference implementation with the demonstration MIDlet running. The faces on the screen illustrate the sound sources. The listener is visible on the center of the screen.

7.2 Implementation

The MIDlet is constructed so that it has four major classes that form the core of the program. These classes are Source that represents a sound source; Spy that represents the user; House that contains the information of the floor plan and geometrical routines; and MansionCanvas that takes care of instantiating all the other important objects, painting the graphics and reacting on user's key presses. Additionally, there are smaller classes that provide necessary auxiliary processing for the main classes. The structure of the program is shown as a class diagram in Figure 25.

Source has a Player instance to take care of the sound playing. A LocationControl is used to update the position of the sound source. Source has also a run method that is called from a separate thread. The run updates the location of the Source repeatedly in a random way. Additionally, Source has a draw method that is called by MansionCanvas.paint when the Source is drawn to the GUI.

Spy represents the user in the virtual world. Spy has a Spectator instance to take care of the room and listener modeling. LocationControl and OrientationControl are used to move the virtual user accordingly when the user moves and a Reverb created by an EffectControl is used to model the reverberation properties of the room. When the user walks to another room the Reverb settings are changed accordingly.

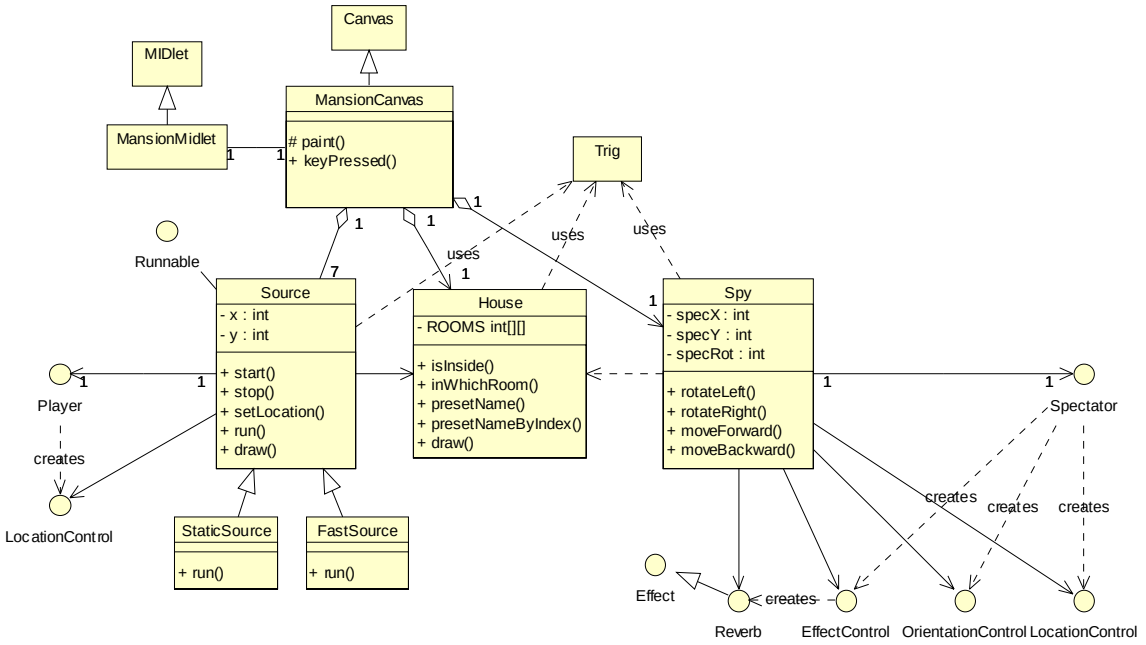

Figure 25. The classes of AAI demonstration MIDlet. Only the most relevant methods and associations are shown.

House has the geometry and reverberation settings of each room in the building stored. House has a draw method that draws the floor plan based on this stored geometry when it is called from MansionCanvas.paint. Additionally, House provides collision detection calculation routines that are used by the Source and Spy to prevent walking through the walls.

MansionCanvas instantiates the House, the Spy, and all the Sources. It also creates separate threads for every Source to run Source's random movement routines. MansionCanvas.paint takes care of painting all the graphics on the screen, but it is assisted by subroutines Source.draw and House.draw that paint their part of the graphics. Furthermore, MansionCanvas reads the user input and redirects the user commands to appropriate objects providing also the source occlusion calculations during the paint routine.

Besides the main classes presented above, helper classes are needed. Trig provides support for trigonometric operations that Source, House and Spy need to use. Sine and cosine are not part of CLDC 1.0 or MIDP 1.0 so those functions are provided by Trig and also coordinate transformations between Cartesian and spherical coordinate systems are there. Besides Trig, Source has two inheritors: StaticSource and FastSource. They extend Source's functionality by overriding the run method. StaticSource does not move at all and FastSource, on the other hand, moves much more than an ordinary Source.

8. Conclusions

The problem addressed in this thesis was how to access and control audio processing features of a modern portable communications device.

As a background, psychoacoustics and its application in virtual acoustics were reviewed. On the other hand, traditional ways to access audio processing features from a computer program were studied and considered. These ways were five different programming interfaces common in the desktop computing nowadays. Moreover, one interface specifically designed mainly for media playback and recording in mobile devices was also studied. This interface formed the basis for the own work done in this thesis.

As the main result, this thesis presented one new interface to control audio processing in a mobile information device. This interface was implemented and a demonstration application was build on top of it.

All the APIs described in this thesis including the novel API were compared. The main found lacks of the novel API could be stated to be the following. It is incapable to control the resource consumption of the audio processing. This feature could be relatively important to have, because we are talking about resource limited mobile devices. It could be easily gained by setting priorities for various sounds. Another lacking relatively important feature is the automatic calculation for the occlusion and obstruction caused by obstacles, such as walls, in the virtual acoustical space. This would need a definition system for the room geometry to the API. That would considerably make the API bigger.

Otherwise, this novel API designed for the mobile world was capable of doing many of the same things as bigger APIs of the desktop computing world. The novel API was designed to be both lightweight and easy to use, and as the matter of fact, the first actual user of the API professed to have learned the API in one hour.

As the interface presented is concentrated on audio signal processing, a possible future work could provide a similar interface for video signal processing.

9. References

[A3D] "*A3D 3.0 API Reference Guide*", Aureal Inc., 2000.

[Blauert 1997] Blauert, "*Spatial hearing: the psychophysics of human sound localization*", Revised Edition, The MIT Press, 1997.

[CLDC 2000] "*CLDC Specification, V1.0a*", <URL: http://jcp.org/aboutJava/communityprocess/final/jsr030/>

[CLDC 2003] "*CLDC Specification, V1.1*", <URL: http://jcp.org/aboutJava/communityprocess/final/jsr139/>

[COM] "*Component Object Model technologies*", www site, Microsoft Corporation, <URL: http://www.microsoft.com/com/>

[Creative] "*Environmental Audio Extensions: EAX 2.0*", version 1.3, Creative Technology Limited, 2001.

[DirectX 9.0 2002] "*DirectX 9.0 Programmer's Reference*", Microsoft Corporation, 2002

[Goldstein 1999] Goldstein, "*Sensation and Perception*", the fifth edition, Brooks/Cole Publishing Company, 1999.

[Hagen, Muschett 2002] Hagén, Muschett, "*Gamer's Guide to 3D sound and reverb APIs*", updated January 8, 2002, <URL: http://3dsoundsurge.com/features/articles/APIs/APIs.html>

[Horstmann, Cornell 1999] Horstmann, Cornell, "*Core Java 1.2 Volume 1-Fundamentals*", 1999.

[Huopaniemi 1999] Huopaniemi, "*Virtual Acoustics and 3-D Sound in Multimedia Signal Processing*". Libella Oy, Espoo, Finland. The report series of Helsinki University of Technology, Laboratory of acoustics and audio signal processing, 1999.

[I3DL2 1999] "*IA-SIG Interactive 3D Audio Rendering Guidelines (Level 2)*", MIDI Manufacturers Association, Sep. 20, 1999. <URL: http://www.iasig.org/wg/closed/3dwg/3dl2v1a.pdf>

[JAVA3D 2002] "*Java 3D 1.3 API Documentation*", 2002. <URL: http://java.sun.com/products/java-media/3D/forDevelopers/J3D_1_3_API/j3dapi/>

[JCP 2001] "*JCP 2: Process Document* ", Sun Microsystems, Inc., 2001. <URL: http://jcp.org/procedures/jcp2/>

[Karjalainen 1999] Karjalainen, "*Kommunikaatioakustiikka*", korjattu esipainos, Libella Oy, Espoo, Finland. The report series of Helsinki University of Technology, Laboratory of acoustics and audio signal processing, 1999.

[Kovach 2000] Kovach, "*Inside Direct3D*", Microsoft Press, Washington 2000

[Kujala, Paavola 2002] Kujala, Paavola, "*Virtual Teleconferencing*", course work for Audio Signal Processing, Helsinki University of Technology, 2002.

[KVM 2001] "*KVM Porting Guide*", Version 1.0.3, Sun Microsystems, Inc., 2001.

[Lindholm, Yellin 1999] Lindholm, Yellin, "*The Java Virtual Machine Specification*", Second Edition, Addison-Wesley, 1999.

[M3D 2003] "*JSR 184; Mobile 3D Graphics API for J2ME*", www page, <URL: http://jcp.org/en/jsr/detail?id=184>

[MaxxVerb 2000] "*Waves' MaxxVerb 'Reverberation' Technology Integrated into Microsoft's DirectX Media Object Architecture in DirectX 8*", Press Release, Waves, Ltd., 2000, <URL: http://www.waves.com/htmls/press/maxxverb_pr.html>.

[MIDP 2000] "*MIDP Specification, V1.0a*", <URL: http://jcp.org/aboutJava/communityprocess/final/jsr037/>

[MIDP 2002] "*MIDP Specification, V2.0*", <URL: http://jcp.org/aboutJava/communityprocess/final/jsr118/>

[Moore 1997] Moore, "*An introduction to the psychology of hearing*", The Fourth Edition, Academic Press, Inc., 1997

[MPEG] "*The MPEG Home Page*", <URL: http://mpeg.telecomitalialab.com/>

[OPENAL] "*OpenAL | Open Source Audio Library*", a www page, <URL: http://www.openal.org/>

[Pereira, Ebrahimi 2002] Pereira, Ebrahimi (editors), "*The MPEG-4 book*", Prentice Hall, 2002.

[Riggs et al 2001] Riggs, Taivalsaari, VandenBrink, "*Programming Wireless Devices with the Java 2 Platform, Micro Edition*", Addison Wesley 2001.

[Savioja 1999] Savioja, "*Modeling Techniques for Virtual Acoustics*", Helsinki University of Technology, Telecommunications Software and Multimedia Laboratory, Report TML-A3, 1999.

<URL: http://www.tml.hut.fi/~las/publications>

[Siemens 2002] "*Siemens scientists develop 3-D streaming speech technology*", press release, <URL: http://www.siemens.com/>, Princeton, New Jersey, February 12, 2002.

[Stautner, Puckette 1982] Stautner, Puckette, "*Designing multichannel reverberators*", Computer Music Journal, 6(1):52-65, 1982.

[Sun 2000] "*Java 2 Platform Micro Edition (J2ME) Technology for Creating Mobile Devices*", Sun Microsystems, Inc., 2000.

[VRML97] "*Information technology -- Computer graphics and image processing -- The Virtual Reality Modeling Language (VRML) -- Part 1: Functional specification and UTF-8 encoding*", ISO/IEC 14772-1, 1998.

[Walsh, Gehringer 2002] Walsh, Gehringer, "*Java 3D API jump-start*", Prentice-Hall, Inc., 2002.

[Zölzer 2002] Zölzer (editor), "*DAFX – Digital Audio Effects*", John Wiley & Sons, Ltd, 2002.

10. Appendix 1: AAI example code

```
//
// creates two sound sources into an alley
//
void createAudioScene() {
    try {
        Player p1 = Manager.createPlayer("http://abc.wav");
        Player p2 = Manager.createPlayer("http://def.wav");
        Spectator s = Manager.createSpectator("");
        p1.realize();
        p2.realize();

        //set the source one 10 meters away to the front
        LocationControl lc_player1;
        if ((lc_player1 =
                (LocationControl)p1.getControl("LocationControl"))
            != null) {
            lc_player1.setUnitsPerMeter(1000);
            lc_player1.setCartesian(0, 0, -10000, 0);
        }

        //set the source two 5 meters away to the right
        LocationControl lc_player2;
        if ((lc_player2 =
                (LocationControl)p2.getControl("LocationControl"))
            != null) {
            lc_player2.setUnitsPerMeter(1000);
            lc_player2.setCartesian(5000, 0, 0, 0);
        }

        // set the listener to origin
        LocationControl lc_spec;
        if ((lc_spec =
                (LocationControl)s.getControl("LocationControl"))
            != null) {
            lc_spec.setUnitsPerMeter(1000);
            lc_spec.setCartesian(0, 0, 0, 0);
        }

        // set global reverb
        EffectControl globalEffectC;
        if ((globalEffectC =
             (EffectControl)s.getControl("EffectControl"))
            != null) {
            Reverb reverb =
                (Reverb)globalEffectC.createEffect("Reverb");
            reverb.setPreset("alley");
            globalEffectC.insertEffect(reverb);
        }

        p1.prefetch();
        p2.prefetch();
        p1.start();
        p2.start();
    } catch (MediaException pe) {
    } catch (IOException ioe) {
    }
}
```