An Embedded Demonstrator for 3D Audio

Ricardo Durón, Björn Johansson
Master of Science Thesis, DTU, 2003

September 30, 2003

Contents

List of Figures

List of Tables

Preface

This thesis is submitted in partial fulfillment of the requirements for the degree of Master of Science at the Technical University of Denmark (DTU), Lyngby, Denmark. Authors are students Ricardo Durón (s011946) and Björn Johansson (s011938). Thesis supervisor is Professor Jan Madsen, Department of Informatics and Mathematical Modeling, DTU and Doctor Harald Gustafsson, Ericsson Mobile Platforms, Lund, Sweden. Project work was done at Ericsson Mobile Platforms, Lund, Sweden between March and October 2003. The thesis is related to the areas of signal processing, algorithm design/implementation, embedded systems and fixed point arithmetic software programming.

Ricardo Durón
s011946

Björn Johansson
s011938

Ericsson Mobile Platforms Lund
30th September 2003

Acknowledgments

We would like to thank Doctor Harald Gustafsson for his encouragement and support throughout the project and for being a good friend, and Professor Jan Madsen for taking part in this thesis and giving us invaluable feedback. We would also like to thank all people at Ericsson Mobile Platform for their positive energy and attitude.

Abstract

With the usage of 3D Audio, it is possible to enhance a non orientated sound wave in such way that it appears oriented within an enclosed space. Although 3D Audio is becoming a standard in the PC gaming world, it is still absent on mobile devices. This thesis will illustrate the development of a fixed point arithmetic 3D Audio engine. The 3D Audio engine is optimized for a Texas Instruments TMS320VC5510-160 DSP on a Pre-OMAP5910 chipset board. The implementation on a DSP platform with limited resources is a step towards the implementation of 3D Audio on mobile devices. The required effects of the 3D Audio renderer are orientation, distance, Doppler and reverberation. Further, the DSP must be able to run the 3D Audio renderer with two playing sources and all effects turned on.

To meet the computational complexity requirements, the 3D Audio structure was designed and optimized to run efficiently on a fixed point architecture. Most of the 3D Audio renderer was developed on the PC platform, and then modified, profiled and optimized to run on the DSP platform. The accuracy of the fixed point 3D Audio renderer is an important part of this thesis. For that reason a floating point implementation of the fixed point 3D Audio renderer was developed in parallel by Ericsson, to compare the accuracy between both versions. The resulting 3D Audio engine was demonstrated on the Pre-OMAP board using its built-in audio codec. The conclusions of this thesis are based on code profiling, to determine the computational complexity of the 3D Audio engine, and Matlab, to verify the accuracy of the fixed point 3D Audio engine.

Acronyms

ANSI American National Standard for Information Systems.

API Application Programming Interface.

CPU Central Processing Unit.

DMA Direct Memory Access.

DSP Digital Signal Processor.

FFT Fast Fourier Transform.

HRF Head Related Filter.

HRTF Head Related Transfer Function.

IFFT Inverse Fast Fourier Transform.

ITD Interaural Time Difference.

LTD Locational Time Delay.

McBSP Multichannel Buffered Serial Port.

TI Texas Instruments.

Mcps Mega Cycles Per Second.

L_b Number of samples to be retrieve from a source

L_f Reverberation Filter Length

L_h HRF Length

L_i Input Length

L_m Sinc Length

L_o Output Length

$LSDV$ ListenerSourceDopplerVector]

N_b Number of buffers per frame

N_c Number of channels

N_d Number of reverberation delay nodes

N_f Number of defined reverberation filters

P_s $Source \rightarrow Next \rightarrow Position$

P_l $Listener \rightarrow Next \rightarrow Position$

V_s $Source \rightarrow Next \rightarrow Velocity$

V_l $Listener \rightarrow Next \rightarrow Velocity$

S_r Sample rate

T_f Frame Period

V_{so} Speed of Sound

Chapter 1

Introduction

This master thesis is written based on an project idea from Ericsson. The project, to create a 3D Audio demonstrator, i.e. to implement 3D Audio algorithms in fixed-point for a Digital Signal Processor (DSP). All parts of a 3D Audio environment should be present; direction of sound, distance, reverberation and Doppler. Further a simple Application Programming Interface (API) shall be created to simplify creation of demos.

The goal of the project is to create an effective functional 3D Audio demonstrator, based on a Texas Instruments (TI) Pre-OMAP DSP platform. The implementation is restricted concerning both Mega Cycles Per Second (Mcps) usage and memory usage.

1.1 Essential Knowledge

- Floating-point is a technique used to represent integer and decimal numbers with decimal numbers, the number is stored with as high precision as possible, while still being able to handle large numbers. This is handled automatically by the math functions.
- Fixed-point is a technique used to represent integer and decimal numbers with just integers. Many platforms are based on integer calculations, with the use of fixed-point it is possible to apply these calculations on decimal values. This handling is accomplished by keeping track of the decimal point.
- Linear interpolation is used to get an approximation of a value. When the start point and the stop point is known, it is possible to approximate a value in between by knowing how far from the start point respectively the stop point the value is.
- Sample rate is the frequency of which the samples were recorded.
- Fast Fourier Transform (FFT) a technique used to transform values from the time domain to the frequency domain.
- Filtering is a technique used to manipulate samples.

- 3D Audio is the actual perception of oriented (directional and distanced) sound.

1.2 Introduction to 3D Audio

3D graphics has improved vastly over a few years due to the rapid development of hardware. The movie and graphics industry have developed 3D graphics into a stunning 3D experience. The focus on 3D graphics has only recently been followed by the development of the next generation audio experience. For many years the principal for high-fidelity sound was stereo sound, this was followed by a technique called surround sound. Surround sound gives the possibility to, by the use of five speakers (one center speaker, two front speakers and two rear speakers), make the sound appear from different directions in the room. This does not give a real 3D Audio experience, since all of this is merely done in the horizontal plane. A correct reconstruction would also include the vertical plane, the entire sound field. By examining different aspects of human physics, i.e. the shape of the ears and the head and how it affects the way we perceive sound, researchers are developing more and more realistic 3D sound models. These models are used with stereo headphones to provide an accurate virtual sound field for the listener.

The technique is also evolving on the speaker side. The aspect of interaction and interference of the right and the left channel must be dealt with (cross talk). 3D Audio is slowly becoming a new standard in PC gaming. Newer game titles use 3D Audio to create a total virtual experience. 3D Audio is also emerging in home entertainment. Several DVD players offer 3D Audio as a complement to Dolby digital 5.1 or other type of surround sound. The DVD hardware uses 3D Audio to create a virtual image of the decoded Dolby digital 5.1 signal. This technology is called Dolby Headphone [1].

1.2.1 3D Audio Overview

3D Audio methods process sound waves prior to they are "displayed" to the human listener to give the impression as if the sounds were coming from a real environment. A sound coming from a position to the left of a listener will cause the sound waves to first reach the left ear and a short while latter the right ear, causing a delay of the right ear signal compared to the left ear signal. The right ear signal will also be attenuated due to the shadowing of the head. There are, of course, other factors that affect the sound waves such as acoustical interaction with the environment and the pinna (the external ear). The shape of the pinna interacts with the sound wave, reinforcing some frequencies and attenuating others depending on the direction of arrival of the sound wave. Human hearing makes use of the factors time delay, amplitude difference and tonal information to determine the location of the sound. These indicators are denoted localization cues. 3D Audio system works by reproducing the sound localization cues at the ears of the listener. By attenuating and reinforcing frequencies, delaying signals, and inserting other effects such as reverberation,the human hearing is fooled to perceive the virtual 3D environment. However, individual human heads/pinnas are all of different shapes and sizes. This gives that for an optimal 3D performance; every individual would need his own set of virtual localization

cues. Since we are used to listen and locate sounds with our own ears, localizing a sound using other people ears could reduce the 3D Audio experience.

1.2.1.1 Acoustic environment modeling

Humans extract a lot of information about the surrounding environment using their hearing. This environment can be modeled into an acoustic environment model. The environment model is a simulation of the acoustical interactions that occur in the real world. The real world can be broken down into three main components: sound sources, acoustic environment and listener.

- Sound source: this is an object in the world that produces sound waves.
- Acoustic environment: Once the sound has been created, it can travel different paths through an environment to the listener and the sound may be modified on the way. The sound may be modified through: absorption (partial or total), reverberation, refraction and scattering.
- The listener: This is an object that receives sound. The listener uses acoustic cues to interpret the sound waves that arrive to its ears and to extract information about the sound sources and the environment.

1.2.2 3D Audio rendering

There are different technologies and effects used to generate 3D Audio, below the methods used to implement the 3D Audio algorithm in this thesis are described.

1.2.2.1 Head Related Transfer Function (HRTF)

To be able to reproduce a 3D environment, Head Related Transfer Function (HRTF) may be used. The HRTFs are used to give impression of that a sound is arriving from a certain horizontal and vertical angle. By placing a sound source at a specified point, at a specific distance (e.g. 1 meter), the HRTF filter set can be created by for all points measuring the sound modification on arrival at the ear channel at both left and right ear. This test is done either by the use of a human test person or by the use of a human head model. The test is performed by placing the test object in center with microphones in the ears and placing a column of speakers in front of the test object. Then a sound is played from each of the speakers in the column and for each sound the result is recored. Thereafter the column is moved one step and the process is repeated until all directions has been measured [4].

1.2.2.2 Interaural Time Difference (ITD)

When a sound source is not placed directly in front of a listener, the sound wave will reach one of the ears earlier than the other. For instance, this means that when a sound source is placed to the left of a listener, the sound will first arrive at the listeners left ear, being slightly delayed at the right ear. (Refer to figure 1.1 for an illustration). This delay is used by the brain to interpret which side of the head the sound source is located and also to determine the vertical direction of sound [3]. When measuring the HRTFs, the measurements will also show a delay between the ears, this delay is separated from the HRTF filter set

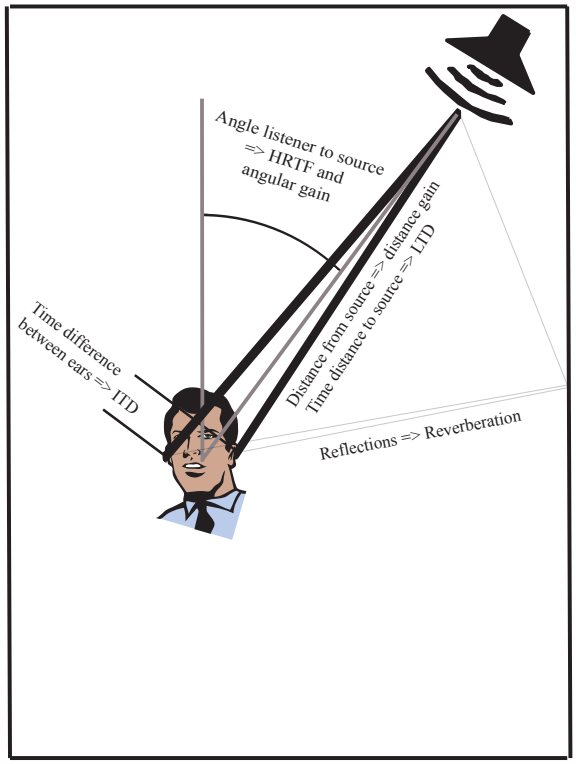

Figure 1.1: This figure illustrates some of the physics used, i.e. ITD, HRTF, LTD, Distance/Gain and Reflection, to create a 3D Audio experience. The ITD and HRTF creates the effect of angle positioning. Distance/Gain gives a source its loudness. The LTD is used to create motion effects and the Reflections are used to give characteristics of, for instance, a room.

and stored as the Interaural Time Difference (ITD) set, the ITD set consists of half the amount of HRTF horizontal filters.

1.2.2.3 Locational Time Delay (LTD)

A person looking at a storm in the distance, will see lightnings before the sound of thunder can be heard. A phenomena like this occurs since the speed of sound is much lower than the speed of light. The time elapsed from the sound is created until it reaches the listener is denoted Locational Time Delay (LTD). In air the speed of sound is approximately 340 m/s, hence a sound generated 400 meters away will arrive to the listener 1.17 seconds after the generation.

1.2.2.4 Loudness

The principal cue for distance is the loudness(gain) of the sound. A source will sound louder when it is closer to the listener. Frequencies may also be partially or totally absorbed by air(depending on humidity), walls or by other objects. Another important cue for distance is the relative loudness of reverberation (see Reverberation in section 1.2.2.5).

The sound sources angle toward the listener is an important fact as well, since a sound source has the option to only send sound waves in a particular direction and to apply different gains the sound depending on the angle. This is called *Angular gain*. The angular gain modifies the loudness of a source depending on which directions it is allowed to produce sound from, and the amount of gain applied to that direction.

1.2.2.5 Reverberation

When a sound source produces a sound, a sound wave expands from the source reaching walls and other objects. The sound wave is reflected and absorbed by walls and other objects. Reverberation is the part of the sound wave that is reflected from objects in the environment in contrast to the direct path sound wave. Depending on the materials in a room, different materials absorbs different amount of sound energy and frequencies, hence the amount of sound reflections will be more or less extensive. Assuming that a sound wave has a direct path to a listener, the listener will first hear the direct sound followed by other reflections from the surrounding environment. This means that the listener perceives the sound wave coming from various directions and at different points in time. Reverberation is an important acoustic phenomenon since it gives useful information about the size and content of the surrounding environment. Small and large rooms can give similar type of reverberation, depending on the contents and materials. Reverberation is also important for establishing distance cues. When the distance between the sound source and the listener is increased, the amount of direct sound decreases. The listener may experience that the sound source vanishes into the background ambient created by all reflected sound waves. The ratio between direct/reverberant sounds can be used as a distance cue since the listener may experience less reverberant sounds as being close and reverberant sounds as being distant.

1.2.2.6 Doppler motion effect

The Doppler effect is the pitch-shifting effect that occurs when there is a relative motion between the sound source and the listener. When an object is approaching the listener, the pitch is higher than the pitch of the resting object. This occurs since, in the time it takes the object to emit one waveform, the object has moved closer to the listener. Thus the emitted wavelength is shorter than normal. Similarly, when the object is moving away from the listener, the pitch is lower than the resting pitch, because the emitted wavelengths are longer than normal. One example from the daily life could be when you are standing still on the sidewalk and an ambulance with its sirens on, passes by. A listener will then hear a clear change of pitch in the sound coming from the siren.

1.3 Summary

There are different technologies and effects used to generate 3D Audio. This chapter described a general overview of the methods used to implement 3D Audio in this thesis. Essential knowledge that helps the reader in understanding this thesis is also introduced.

Chapter 2

Development and Design

This thesis consists of a fixed point implementation of a 3D Audio engine. The design of the target environment is a C55x DSP, running at 160 MHz. The development phase, denotes the creation of a working 3D Audio engine without reverberation. This phase was developed on the PC platform. A new floating point version of the fixed point 3D Audio engine was coded in parallel by the supervisor of this thesis. This new floating point code was implemented as similar as possible to the fixed point version and was used to verify the accuracy of the fixed point 3D Audio engine.

On the DSP platform, the working 3D Audio engine was modified to run on both platforms. The cross platform code could be optimized on the DSP and run, be debugged or compared to the floating point code on the PC. This was necessary to verify the accuracy of the 3D Audio engine after optimization modifications. The reverberation block was developed directly on the DSP platform, due to the computational complexity of available FFT functions. These functions were developed for debugging purposes and are not optimized for the PC platform. The FFT functions are mapped to assembly functions on the DSP platform making it possible to run the 3D Audio engine with reverberation in real-time. To run the 3D Audio engine in real-time, sound is sampled from the DSP board's line in jack, processed by the DSP and sent out through the line out jack.

2.1 The Platforms

The platforms used in this thesis are two windows based PCs and one Pre-OMAP board. The development started on the PCs due to its "infinite" resources (compared to a DSP) and due the lack of a DSP (the DSP was not available until the fourth month). It was decided that the fixed point 3D Audio engine should be completed on the PC platform and then modified and optimized to fit on the DSP platform. However, due to the complexity of the used FFT functions, the reverberation block was developed using both platforms. Cross platform code was required, since all verifications of the 3D Audio engine were made comparing the fixed point and floating point versions of the code on the PC. Section 5.1 describes some of the issues encountered, when making a cross-platform (PC and DSP) version of the 3D Audio engine.

2.1.1 PC Platform

The Windows based PCs are one Intel Pentium III 650 MHz with 256 MB RAM, and one Intel Pentium IV 2.4 GHz with 256 MB RAM. The 3D Audio engine was initially implemented on a the PC platform using Microsoft's Visual Studio 6. The intrinsic operators, DSP operations optimized to run on a DSP, are mapped to various functions on the PC platform. These functions are slow on the PC platform, using a large amount of CPU cycles. To be able to communicate with the sound card on the PC, an open-source audio device library called portaudio was used [7].

2.1.2 DSP Platform

The DSP used in this thesis is a Texas Instruments TMS320VC5510-160 running at 160MHz with dual multipliers (up to 320 million multiply-accumulates per second (MMACS)). It has a 256 KBytes program/constants SARAM(Single Access RAM) and a 64 KBytes DARAM (Dual Access RAM) for stack and heap allocations. The program/constants memory is reserved to program instructions and constants. The stack, system stack and heap are fitted into the 64 KBytes memory. The DSP sits on a Pre-OMAP board. This board is a pre-release of Texas Instruments OMAP5910 chipset. The Pre-OMAP board integrates a TMS320C55x DSP core with a TI enhanced ARM925. The stereo codec interface uses a Texas Instruments TLC320AD77C, 24 bits, 96kHz stereo audio codec. Codec input is accessible through a microphone stereo jack, or a line input stereo jack [6]. The Codec is accessible via one of the Multichannel Buffered Serial Port (McBSP), possibilities exists to be able to decide how the long the McBSP buffer will be. It is possible to get interrupts when the buffer is filled, but it is also possible to use Direct Memory Access (DMA) to transfer data from the McBSP to the memory, and after a number of complete transfers get an interrupt from the DMA controller.

The developing tool on the DSP platform is Code Composer Studio 2.2. Code composer studio, runs on the PC and communicates with the Pre-OMAP board. Code composer may also run in simulations mode, i.e. it does not need to communicate with an external board and simulates the execution of a program using the PC's CPU. During this thesis, only one Pre-OMAP board was available so the Pentium IV computer was set to use Code Composer Studio to simulate the Pre-OMAP board.

2.2 Requirements

This master thesis project will develop a 3D Audio renderer suitable for the limited execution environment of the DSP platform. The 3D Audio renderer must include the following functionality:

- Head Related Transfer Function(HRTF) filtering.
- Distance effect.
- Doppler effect.
- Reverberation effect.

A simple API should be implemented to control the 3D Audio renderer, to simplify developing of demonstrations. Furthermore, the implementation must also follow the following guidelines:

- Must be written in American National Standard for Information Systems (ANSI) C [2] to facilitate porting to the DSP platform.
- Must support high/low quality (more/less complexity).
- Must be programed using only fixed point arithmetics.
- Computational complexity must be kept at a low level.
- The 3D Audio Engine, with two playing sources, must run in real-time on the target environment.

2.3 Basics

For the readers convenience, some basic techniques used throughout the thesis are described in the following sub sections.

2.3.1 Effecient Code Construction

Efficient code is not always easy to make. An efficient solution on a PC may be very inefficient on, for instance, an embedded system. Different optimizations produces different results on different systems. An efficient code also considers other aspects such as amount of cache memory, hardware tweaks, etc. In some extreme cases, writing into an array in a specific order, may be enough to get different results due to compiling techniques and memory management. To make the code more efficient, one must consider the computational complexity of the code. Considering this may be very difficult without any actual code. One way to decrease the computational complexity when using platforms that lack floating-point hardware support, is to implement the entire solution using fixed-point operations.

2.3.1.1 Fixed-point Operations and Problems

There is a big difference between the worlds of fixed- and floating-point. Floating-point operations provide developers with great accuracy, but at a high price: computational complexity. Systems with floating-point hardware support, for example a PC, are often optimized and can calculate these operations as fast as fixed-point operations. On the other hand, calculating floating-point operations in systems without this kind of hardware support, would be many times more expensive. In these kinds of systems, developers may choose to only use fixed-point operations to create a more efficient program. This approach uses considerably less computational power, but it gives developers challenges such as range/resolution, accuracy and overflow/underflow problems.

Table 2.1: Range in a 3D-world. The table shows the maximum size the world may have using 16/32 bit fixed-point variables at a desired resolution. If every step in the 3D-world corresponds to 1mm, the maximum world size would be 4.2 million meters using a 32-bit fixed-point variable.

Resolution	*Maximum range in meters with 16-bit integer*	*Maximum range in meters with 32-bit integer*
1 meter	65 536	4 294 967 296
1 decimeter	6 553	429 496 729
1 centimeter	655	42 949 672
1 millimeter	65	4 294 967

2.3.1.1.1 Variables Range and Resolution The number of bits (bit-depth) of a fixed-point variable determines the maximum value it can obtain. The more bits a variable use the greater value it can store. One of the more common problems with fixed-point variables is the limited range. A 16-bit integer has a range of -32 768 to +32 767 i.e. 65 536 steps. If every step measures one meter in the real world, the range would be 65 536 meters. The range would perhaps be enough, but the resolution would be useless. Table 2.1 shows the range/resolution of a 3D-world that can be achieved by using different bit-depths in fixed-point variables. Different systems are optimized for different bit-depths. Usually, operations on a 32-bit variable demands more computational complexity than a 16-bit variable. A developer needs to analyze the code and conclude what kind of bit-depth is needed to satisfy range/resolution and computational complexity demands.

2.3.1.1.2 Accuracy Another limitation is that fixed-point variables can only store integer values i.e. 1, 2, 3. Fixed-point variables cannot store floating-point values such as 0.5, 1.2, or 1.656, which are needed in common calculations. There is of course several ways around this problem. The method used throughout this essay is the "Q-format". The Q-format makes it possible to represent a quantized floating-point value with a fixed-point value. The annotation for this format is the symbol "Q" followed by the number of bits used for storing the decimal value. The bits of a fixed-point variable must be sufficient to describe both decimals and integer part of the floating-point value. A 16-bit signed integer may accommodate a Q15-format. To represent the value 0.75 in Q15-format simply multiply: $0.75 \cdot 2^{15} = 24\ 575$. The same value represented in Q10-format gives: $0.75 \cdot 2^{10} = 768$. Using a higher Q-number gives a better resolution and accuracy. A Q-format with N-bits defines a resolution corresponding to 2^{-N}. Please refer to table 2.2 for a comparison between different Q-formats.

Note that overflow/underflow problems are originated by choosing an incorrect Q-format resolution. Accuracy is very important, the more accurate calculations are, the higher quality of the output.

2.3.1.1.3 Overflow/underflow Most developers are familiar with the terms overflow/underflow. Overflow occurs as a result of an operation that produces a magnitude larger than the given range. Underflow occurs as a result of an operation that produces a magnitude smaller than the given range. Most systems do not handle overflows/underflow automatically, but some systems prevent over-

Table 2.2: Q-format accuracy. The table shows the accuracy of the Q-format. The first row shows the actual floating-point results, the remaining rows show different resolutions of the Q-format.

Resolution	*Multiplication* $0.75 \cdot 35$	*Multiplication* $0.123456 \cdot 50$	*Multiplication* $0.123456789 \cdot 50$
Float	26.25	6.1728	4.074074037
Q31	26.25	6.1728	4.074074028
Q15	26.25	6.1724	4.073763237
Q7	26.25	5.8594	3.867187500

flow/underflow from occurring by saturating when needed. "Wrap-around" is a term used when no saturation occurred and a variable flips from it's maximum value to it's minimum value (or the other way around) i.e. if the value 33 000 is stored in a signed 16-bit integer, the resulting value would become: (33000-32767)+(-32768)= -32 535, since the maximum value is 32 767 and the minimum value is -32 768. To discover wrap-around problems, developers must analyze their code keeping in mind the number of bits used and the maximum number of bits allowed. A simple example of the wrap around problem would be multiplying two 16-bit variables and saving the result as a 16 bit variable. This multiplication creates a wrap around since multiplying two 16-bit variables may give values larger than 16-bits i.e. $2^4 \cdot 2^{13} = 2^{17}$. There is the option of choosing a higher bit-depth, but using a 16-bit integer is less complex than using, for instance, a 32-bit integer. A remedy to this kind of problem is a method called Binary Floating point. A value is represented by $s \cdot M \cdot 2^E$, where s is a sign bit for the mantissa, M and where E is an exponent. This method allows a 16-bit integer, with the help of a low depth integer, to represent values that normally only can be stored using higher bit-depths. The exponent, E, is a binary integer in some interval, typically eight bits. The mantissa is a number in the interval [1/2, 1]. However, the value 1/2 cannot be represented using integer variables. A way around this problem is to use a Q-format to represent the interval [1/2, 1]. Using the Q-15 format the interval would instead be [16386, 32767]. The Q-format is chosen depending on the bit-depth of the mantissa i.e. always choose the highest Q-format that the mantissa may accommodate to acquire the best resolution. Using the binary floating point method, the value 33 000 can be represented as: 16 $500 \cdot 2^1$. Overflow may also occur when setting the accuracy too high in the Q-format. By having high accuracy, fewer bits are available to storing the integer value. The correct representation of a value using the Q-format depends much on how the number of bits are assigned to the integer and decimal parts. A larger amount of decimal-bits increases the accuracy of the represented value. On the other hand, if the bits assigned to represent the integer part are not enough, a wrap-around will occur resulting in a completely wrong value. Table 2.3 shows what happens when choosing an accuracy that is too high. Four different values are going to be represented using different Q-format. The integer part of the first value 2 047.12,needs at least 11-bits to be represented (fewer bits gives overflow), the integer part of 127.12 needs 7-bits and the integer part of 1.12 needs 1-bit.

Correcting overflow/underflow problems cost computational power. By profiling the code, i.e. running the program and gathering statistics about functions

Table 2.3: Overflow problem using the Q-format with 16 bit integer. Every column has its own Q-format chosen by looking at the integer part of a selected row, and taking the least number of bits needed to represent the integer part of this floating-point value.

Floating point value	*Q14* *1-bit integer* *14-bits decimal*	*Q8* *7-bits integer* *8-bits decimal*	*Q4* *11-bit integer* *4-bits decimal*
2 047.12	Overflow	Overflow	32 753
127.12	Overflow	32 542	21 521
1.12	18 350	286	17

or lines in the code, developers see the computational complexity of selected areas in the code. This gives developers a better view of the code and helps them conclude when it is better to correct overflow/underflow problems at the same bit-depth and when it is better to simply increase the bit-depth of variables.

2.3.1.2 Fixed point Optimizations

The target implementation of 3D Audio is on an embedded system, a DSP TMS320C55x. This type of DSP does not have hardware support for floating point operations. The code is restricted to use integers to fulfill the demands the of a low computational complexity. The TMS320C55x can handle a data word size of 16-bit, but does all calculations on a 40-bit ALU. This means that greater performance will be achieved by keeping variables as 16-bit data words. Division, as in most architectures, is to be avoided if possible due to its complexity. Shifting may be used instead of division/multiplication if the nominator is a power of two. The TMS320C55x can handle shifts simultaneously with other other arithmetics in one cycle which makes shifting far more effective than division/multiplication.

There are other functions even more complex than a division for instance arcus tangent, square root, power, etc. These functions are used in the geometric calculations. Approximating these kinds of functions results in sufficient accuracy and reduced computational complexity. For more information about the approximated functions refer to section 3.2.11.

2.4 3D World Model

The 3D World model was constructed to fit an fixed point implementation considering the limitations of 16-bit and 32-bit values. The 3D world model consists of one listener, several sound sources, several specified environments (scenes) and the 3D positioning space (the 3D world).

2.4.1 The 3D World

One of the major issues was the space of the 3D world. The area and geometry of the 3D-world must be decided before commencing with the 3D Audio engine. From the starting point, the center of the 3D space, the world can be +/- MAX_INTEGER_VALUE for all axis. Since the world is build upon integers,

no fractions are available. Every step in the integer distance must correspond to a certain unit with enough precision to satisfy the calculations of 3D audio. For instance, an object in the x-axis of the world is moved one step from the distance 256 to 257. If the unit of the world were in meters, the object would have moved 1 meter. If the unit of the world is chosen to 1/64 meter, the object would have moved about 1,6 cm. Most calculations in the 3D Audio engine are based on the position vectors of different objects. This means that the smallest step in the 3D world must have enough precision to give small distortion errors in the rest of the calculations. A step in the 3D Audio world was defined to 1/1024 meter, about 1 mm. The maximum error would be 1 mm in every axis resulting in the distance: $\sqrt{0.001^2 + 0.001^2 + 0.001^2} \approx 0.002$ meter, about 2 mm. This error is acceptable, since the HRTF and ITD values are measured 1 meter outside a persons head, a listener will never experience the sound closer than 1 meter, making the maximum error about $2 \cdot 10^{-3}$. The range of the world using 16-bit variables with this kind of resolution would be $2^{16}/1024 = 64$ meters, which is by far not enough. Using 32-bit variables gives a range of $2^{31}/1024 = 2097152$ meters about 2100 kilometers, or 1050 km in every direction from the center.

2.4.2 Scenes

The 3D world may be divided into areas, called scenes. These areas may have different characteristics, like different amounts of reverberation and different amount of distance gain. When a new scene is used, the listener experiences that he has entered a new room, due to the scenes different characteristics. This thesis does not implement scene switching, although the scene functionality is present. The scene implementation includes distance vectors that describe how the amplitude of a sound is altered by distance. The implementation of reverberation is not tied to a scene but to the entire 3D world.

2.4.3 Listener

The Listener is an object that need to keep track of its orientation. If a listener is upside down, the world is experienced differently than if he were standing straight up. The listener may also tilt his head, moving the head without turning it around. These alterations of the listener does not involve a change of position, but changes the perception of the 3D world. To handle these kinds of alteration, a *Up* vector was constructed. The up vector handles situations like tilting of the head or when the listener is upside down. The listener has also the ability to face a source or to turn away from the source. To handle this types of situations a *Front* vector is required. The front describes the direction the listener is looking in.

2.4.4 Sound Source

A sound source has the option to spread their sound omnidirectional or directional. Omnidirectional sound is created by, for instance, a car crash on the free way. All sound waves spread in all directions. A directional sound could be described as a person "P" standing with his back in front of a thick concrete wall, talking. A person on the same side of the wall as "P", hears what "P" is saying. A third person is situated behind the wall. He cannot hear what "P" is saying.

If "P"'s back is as close to the wall as possible, he would produce sound in the range 0 to ± 90 degrees. If "P" then picks up a pipe and starts talking through it, he would produce sound in a even smaller range. To be able to render this, the sound source has an angular vector and a *Front* vector. The front vector lets the source face a listener or turn away from him. The angular vector tells the source in which angles, seen from the source's front vector, it may produce sound from and what angular gain it should use in the particular direction. This enables a source to spread sound omnidirectional (0 to 360 degrees), in a specified direction ($angle_1$ to $angle_2$) and in a specified direction using different gains depending on the direction ($angle_1$ to $angle_2$ use $gain_1$; $angle_3$ to $angle_4$ use $gain_2$;...). Note that the sound sources angular vector contains angles in the range $0 \leq X \leq 180$, since the sound source is symmetrical. This means that if a user sets the sound direction of the source from 10 to 20 degrees, the sound source will also produce sounds from 350 to 340 degrees.

2.5 Summary

This chapter introduced the different platforms used by this thesis. The requirements of this thesis and the flow used to develop 3D Audio were described. Knowledge on common fixed point problems and fixed point optimizations were presented. Furthermore, the design of the 3D Audio world and the different elements in the world like scenes, listener and sources were also described.

Chapter 3

3D Audio Render Implementation

This chapter will describe the render procedure of the 3D Audio engine. An overview of the structure and flow of the program will first be described, followed by the different blocks of 3D Audio.
The render routine is the heart of the 3D Audio engine, meaning that a considerable amount of time was spent to obtain an efficient approach. One of the requirements was to make the renderer flexible, in the sense that the renderer could be set to render in high quality or lower quality, varying the computational complexity.

3.1 Structures and Flows

The 3D Audio renderer is a routine called A3D_execute, that loads, modifies and returns a set of samples to create the 3D Audio experience. To reduce the computational complexity, all of A3D_execute does not need to be executed every update. The part that needs to be executed every update, i.e. each time an out-buffer is produced, is called *Buffer Update Routine*. The Buffer Update routine approximates and applies values and filters in order to modify

Table 3.1: Notations and symbols.

Symbol	*Description*
L_i	denotes the input length.
L_o	denotes the output length.
L_m	denotes the Sinc length.
L_h	denotes the HRF filter length.
L_f	denotes the reverberation filter length.
L_b	denotes the number of samples to be retrieved from a source.
N_d	denotes the number of delay nodes.
N_f	denotes the number of defined reverberation filters.
N_c	denotes the number of channels.
S_r	denotes the sample rate.

the incoming samples, see section 3.1.2 for more details. The values used by the Buffer Update routine are values that are calculated by the other part of A3D_execute, called *Frame Update Routine*. The Frame Update routine does not need to be executed every time A3D_execute runs and reduces the average complexity of A3D_execute. Since the Frame Update routine updates values such as positions and angles, the quality of the output depends on how often the Frame Update routine is executed. More on the Frame Update routine in section 3.1.1.

The frequency of the Buffer Update's execution depends on the outgoing buffer size, i.e. the number of samples rendered by the 3D Audio engine. If the outgoing buffer size is set to 48000 samples, assuming that the sample rate is set to 48 kHz, the Buffer Update routine will execute 10 times per second. Using a more realistic buffer size, 2 - 20 ms, would yield execution frequencies of 500 to 50 times per second.

The implementation split into two main parts offers one big advantage. When using the 3D Audio engine together with, for instance, a 3D graphics engine, the Frame Update may be set to the same frequency of execution as the 3D graphics frame render routine. If the 3D graphics engine generates 60 frames per second, the Frame Update routine will also be executed 60 times per second, about every 16 ms. Assuming that the Buffer Update routine is set to execute every 2 ms, the Frame Update would run once every eight Buffer Updates, thus reducing the complexity without compromising the quality. A block diagram of A3D_execute is available in figure 3.1.

3.1.1 Frame Update Routine

The Frame Update routine calculates and stores values for both listener and audio sources. Every source and listener in the 3D Audio world has three different structures: *Next, Current* and *Previous*. These structures are the key feature why the Frame Update routine may be executed with less frequency than the Buffer Update routine. The Next-, Current- and Previous structures stores different values. All calculated values in the Frame Update routine are stored in the Next structure. The Next structure defines the point where a source or listener are moving towards, i.e. all values stored in the Next structure are future values. Before the Frame update starts calculating the Next-values, the Next- and Previous structures are flipped, i.e. Next becomes Previous and Previous becomes Next. By doing so, the values of the previous Frame Update are always stored in the Previous structure, and the newly calculated values of the executing Frame Update are stored in the Next structure. This approach opens an opportunity for the Buffer Update routine to linear interpolate the Next- and Previous parameters allowing the Frame Update to run less frequently if needed. The less frequent the Frame Update routine is executed, the older are the Previous values. This results in less accuracy when interpolating between previous and next values. See figure 3.2 for an illustration on how the quality of the output may be affected by the number of calls to the Frame Update routine.

3.1.1.1 Frame Update Routine Flow

An overview of the Frame Update Routine is illustrated in figure 3.3. The Frame Update routine starts by updating the listener structure if the application has

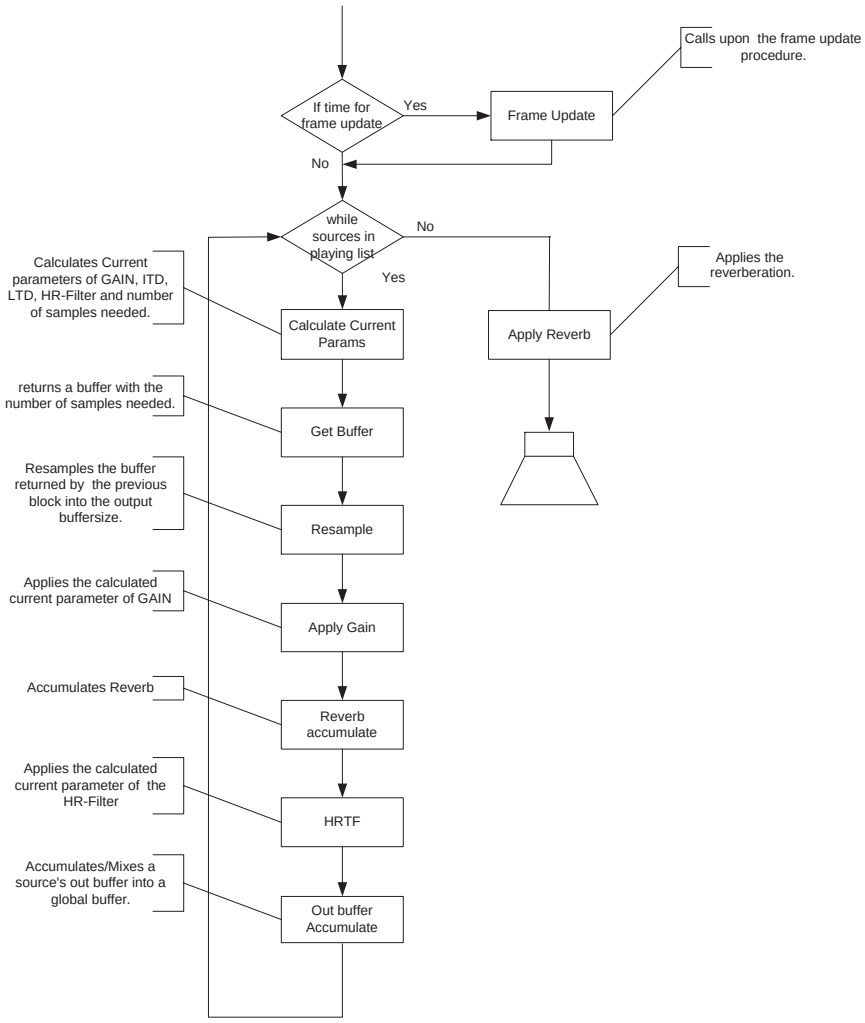

Figure 3.1: A3D_execute. The figure shows the flow of A3D_execute. The values calculated by the frame update are used by the rest of the blocks. The rest of the blocks are the Buffer Update Routine i.e. the part of A3D_execute that must execute for every out-buffer. The precision of current parameters in the Buffer Update part of A3D_execute depends on how often the Frame Update Routine is executed. The samples that come out of A3D_execute need only to be converted through a D/A converter and then listened to using an amplifier and a speaker.

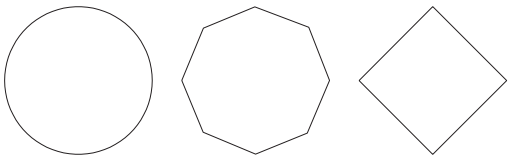

Figure 3.2: Frame Update Precision. The figures illustrates how a listener experiences a source spinning around his head in a circular trajectory with a different number of calls to the Frame Update routine. The figure to the left represents an "infinite" number of calls calls to the Frame Update routine. The figure in the middle represents the same trajectory, using only eight Frame Updates and the figure to the right uses only four Frame Updates. The more frequent the Frame Update Routine is executed, the more precise the output will be.

changed any value corresponding to the listener. If an updated is needed, the positioning, front, up and velocity vectors are updated and listener projections in the 3D-world are calculated. If no changes have been made to a source after the previous frame update, the Previous values are copied into the Next structure, so that the Buffer Update routine interprets this as the listener not being in motion. Once the listener is successfully updated, the Frame Update routine starts updating the sources in both playing and muted lists if their values have been altered via the API. Again, if no changes have been made to a source since the previous frame update, the Previous values are copied into the Next structure, so that the Buffer Update routine interprets this as the source is not moving.

All sources in both playing and muted lists are updated using the source update procedure. An overview of the update source procedure is illustrated in figure 3.4. The source update procedure starts by updating the positioning, front and velocity vectors of a source and calculating the distance to the listener. If the distance to the listener is bigger than the largest value in the distance vector, the source is moved to the muted list. The distance vector, is a vector set via the API, used to calculate the gain of a source. This vector is a set of distances paired with gains. The largest value of the distance vector is the largest distance that has a gain attached to it. If the distance from a source is greater than the largest value of the distance vector, it is moved to the muted list and denoted as *not active* to gain performance. If a source is *active*, the Gain, LTD, ITD and HR-Filter values are calculated and stored in the Next structure. There are of course many more calculated values in the update source procedure routine, but the mentioned values are the ones used by the Buffer Update routine.

3.1.2 Buffer Update Routine

The Buffer Update routine processes all playing sources and modifies them using gain, ITD, HR-Filter and in some cases Doppler and reverberation. A set of samples are loaded from a source and modified using approximations of values calculated in the Frame Update routine. The approximation used throughout the Buffer Update routine, is a two point interpolation of the values between the Next and Previous structures. The calculated approximations are saved into

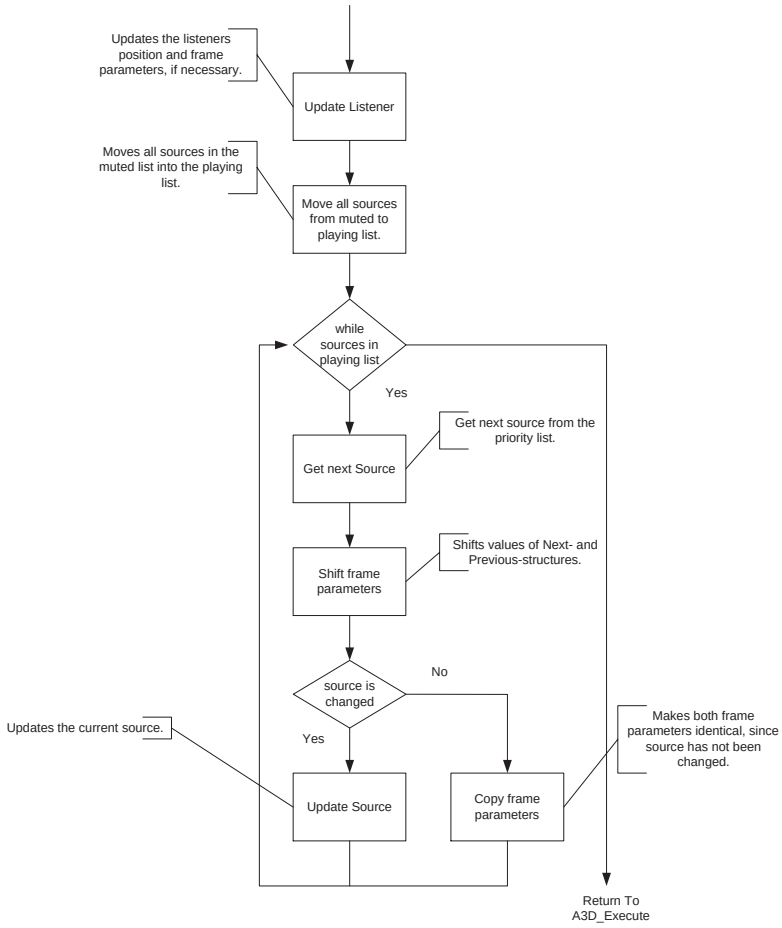

Figure 3.3: The figure shows the flow of Frame Update Routine. After updating the listener, the Frame Update routine starts updating sources in both muted and playing lists, if necessary.

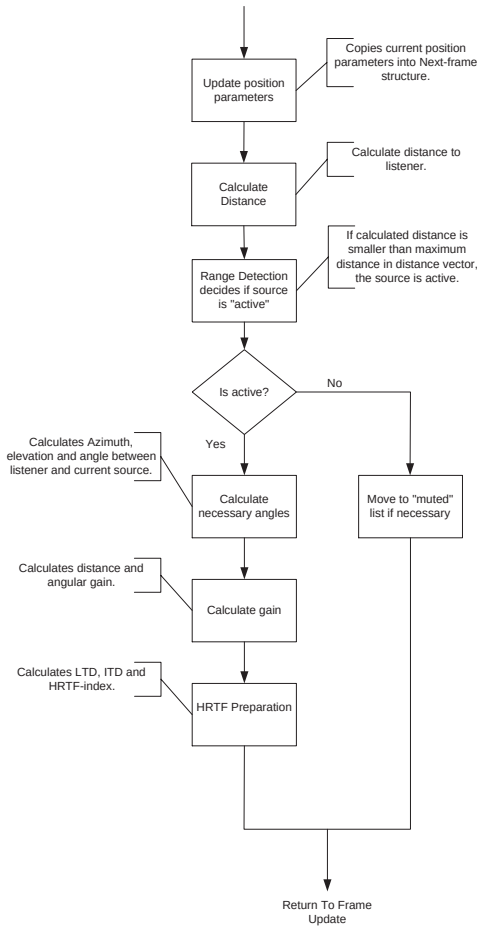

Figure 3.4: The figure shows the flow of the Source Update Procedure. A source is moved to the muted list and labeled as *not active* if the distance to the source is greater than the largest value in the distance vector. If a source is active, Gain, LTD, ITD and HR-Filter values are calculated and stored in the Next structure.

the Current structure as seen below:

$$
\begin{aligned}
DivisionGain &= \frac{32767}{max\_steps} \\
Current &= ((current\_step \cdot Next) + \\
&\quad ((max\_steps - current\_step) \cdot Previous)) \\
&\quad \cdot DivisionGain
\end{aligned}
$$

Note that DivisionGain is only computed once in the initialization procedure of the 3D Audio engine. There are *maxsteps* steps in the buffer update routine. *Currentstep* is how many steps have been executed so far.

The main task of the Buffer Update routine is to modify the incoming samples of all playing sources using the parameters in their respective Current structure. The Buffer Update routine also accumulates all output samples into one common stereo buffer, mixing all playing sources.

3.1.2.1 Buffer Update Routine Flow

A schematic overview is shown in figure 3.5. A set of samples (in mono) are first modified by the approximated gain. Depending on the sources distance to the listener and the angle from the source to the listener, the samples are made louder or quieter. The block of Doppler and ITD, re-samples the samples modifying the pitch depending if the source is moving towards or away from the listener and inserts the Interaural Time Delay (ITD). At this stage the source mono channel has been separated into three channels, two channels(left and right) have been created to obtain the ITD effect and one more channel is created for the reverberation. The next processing step is to apply the HR-Filter values to the left and Right channels, altering the samples so that the output is experienced as external the listener's head. Please note that the third channel has only been altered by the Doppler effect, and does not contain any modifications done by gain, ITD or the HR-Filter. This procedure is made for every source, accumulating the modified left and right channels into a two channel mixed buffer and the third channel into a reverberation buffer. After all sources have been processed, the reverberation is mixed into left and right buffers. Using a D/A, a stereo signal is created that can be played using an amplifier and stereo speakers.

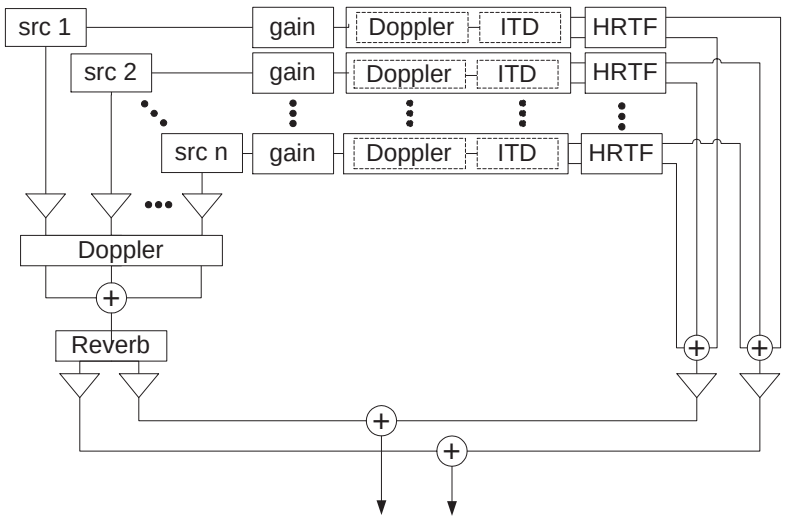

Figure 3.5: Schematic overview of the buffer update routine. The overview shows that there may be a number of sound sources producing samples in mono. The signal will be processed in two ways. The first is performed individual for each source, where the signal is mono until Doppler and ITD has been applied, thereafter may the signal differ between right and left channel and the HRTF will be applied to both channels, the result will then be mixed with the out signals from all sources. The other processing of the signal is by first applying Doppler and thereafter to mix the signals from all sources, and to apply reverberation.

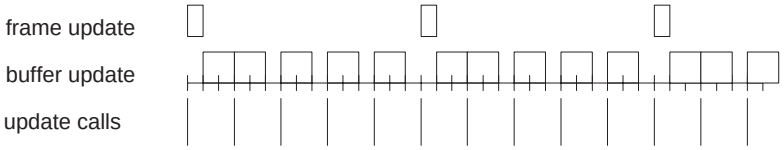

Figure 3.6: This figure illustrates the execution of the frame- and buffer update routines. Every time an update call is thrown, a buffer update is executed. The frame update does not need to be executed every update call. The update frequency of the frame update is set via the API in the initialization phase.

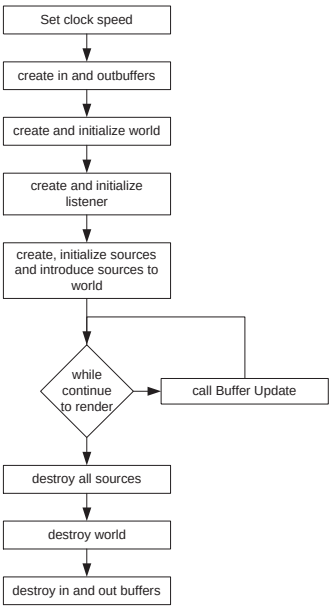

Figure 3.7: Main function flow. The main functions general flow is first to create and initialize everything needed, then to execute the actual rendering and thereafter to clean up.

3.1.3 Test application

The main function handles calls to create and destroy world, listener and sources. It also handles execution loop, see figure 3.7. A number of initialization functions handles resetting and setting of variables at start and during execution. The initialization of World calls most of them, see figure 3.8.

To be able to get a good program flow, it is important to have good data structures. Most of the data structures depends on each other, e.g. a source can not exist without a world, nor can a listener. The structures for Listener, Source, Scene have been created in such a way that they reflect the physical elements as far as needed. List structures to keep track of sources have been created to make it as easy and fast as possible to parse and to minimize the amount of checking needed. Reverberation structures are written in such way that there might be some memory allocated that is not for the moment filled, though minimized the amount of copying needed. See figure 3.9.

3.1.3.1 A3D_World

A3D_World is the overall structure containing everything related to sources, environments and listeners. This structure is basically the one thing that always needs to be one of the arguments to all rendering functions. It is possible to add, change and remove sources, listener and environment in the world.

- Reverberation is the structure containing everything related to reverberation. It contains all sample buffer for the overlap and save filtering which

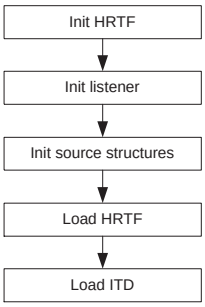

Figure 3.8: Init World flow. When initializing the world, everything connected to the world will also be initialized.

limits how long delay that is possible.

  - ReverbDelayBuffer is the structure containing data necessary to create the reverberation delay effect. The ReverbDelayBuffer has a pointer to the newest object in the circular delayList which consists of nodes of type ReverbDelayNode.
  - ReverbDelayBuffer is the structure containing the FFT values of the input data. It keeps knowledge about the normal of the FFT values and the amount of data stored.
  - ReverbBuffer is the structure containing in- and out-data buffering. It has a first and a last pointer, to the same simple linked list, it also holds the number of samples collected.
  - ReverbBufferNode is the structure containing the actual samples for in- and out-data buffering. It contains a start offset so that it is possible to just retrieve a chunk of the data.
  - FilterH is the structure containing the FFT values of the filters and their norms. This is used when applying the reverberation.

- HRTF is the structure containing the specified loaded HRFs, pointers to specific intervals within the filters and the filter order.
- FrameData is the structure containing orientation of the world.
- Sources is the structure containing number of sources introduces to the world and the sources.
  - SourceList contains the Playinglist which is a linked list with priorities, the Muted/Paused/Dissabled list which are simple linked lists.
- Listener is the structure containing all information about the listener, e.g. position, which HRF to use, and direction.
- Scene is the structure containing environment specific information.
- SourceData is the structure containing all information about a source, e.g. position, volume, type of source and where to read data from.

- SourceBuffer is the structure containing data used in rendering sound from a source.
- SourceRenderParam is the structure containing parameters used in rendering sound from a source.
- CommonSourceData is the structure containing work buffers used for each playing source in Buffer Update.
- SourceTimeEvent is the structure containing loop information for a specific source such as start point, stop point and number of times to loop.

3.1.4 Buffers

A number of sample buffers are used in the program, there are both buffers that are temporary and buffers that are persistent. Buffers are used to be able to just retrieve a small chunk of a large block, to be able to retrieve a large chunk from small blocks, to retrieve data and to process it and to store data e.g. between Buffer Updates.

3.1.4.1 Buffer Update

Most of the buffer processing takes place in the Buffer Update function. To give a better overview of the data flow, the Buffer Update will be simplified, so that LTD (Doppler), ITD, HRTF and gain blocks are inserted into a Source Buffer Update. The data flow between buffers is shown in figure 3.10 and the execution flow is shown in figure 3.5. The Source Buffer Update describes where the buffers of a source are modified. The Source Buffer Update starts by retrieving L_i samples into an input source buffer. The number of samples, L_i, depends on the ITD and LTD values.

The input source buffer is modified by distance and angular gain and sent to a re-sample function (LTD/Doppler block). Notice that this block is illustrated twice, as the Doppler block before Reverberation and as Resample in the Doppler (LTD and ITD) block, in figure 3.10. The input buffer is resampled from a buffer length that changes every Source Buffer Update, into a fix buffer length. The input buffer is also split into three channels (three buffers). The ITD is only applied to the first two channels, to create the delay between left and right ear. The samples in these two channels are transformed by the HRTF filter and stored in the output buffer of the source.

The third channel that is created by the resample function is used by the Reverberation accumulate block. The Reverberation accumulate block accumulates the third channel of all playing sources into a new buffer. The accumulated buffer is sent to the Reverberation block. In the Reverberation block the accumulated buffer will be stored in the Reverberation in-buffer, from where it is possible to retrieve smaller blocks of data. As long as the reverberation in-buffer contains more then or equal to L_i samples, L_i samples will be retrieved, zero padded and frequency transformed via FFT. The result of the FFT is sent to the delay buffer, where N_d delay blocks is stored, these blocks are filtered, summarized and then time transformed via Inverse Fast Fourier Transform (IFFT). The result of the IFFT is then stored in the reverberation out-buffer, from where it is possible to retrieve blocks of desired length.

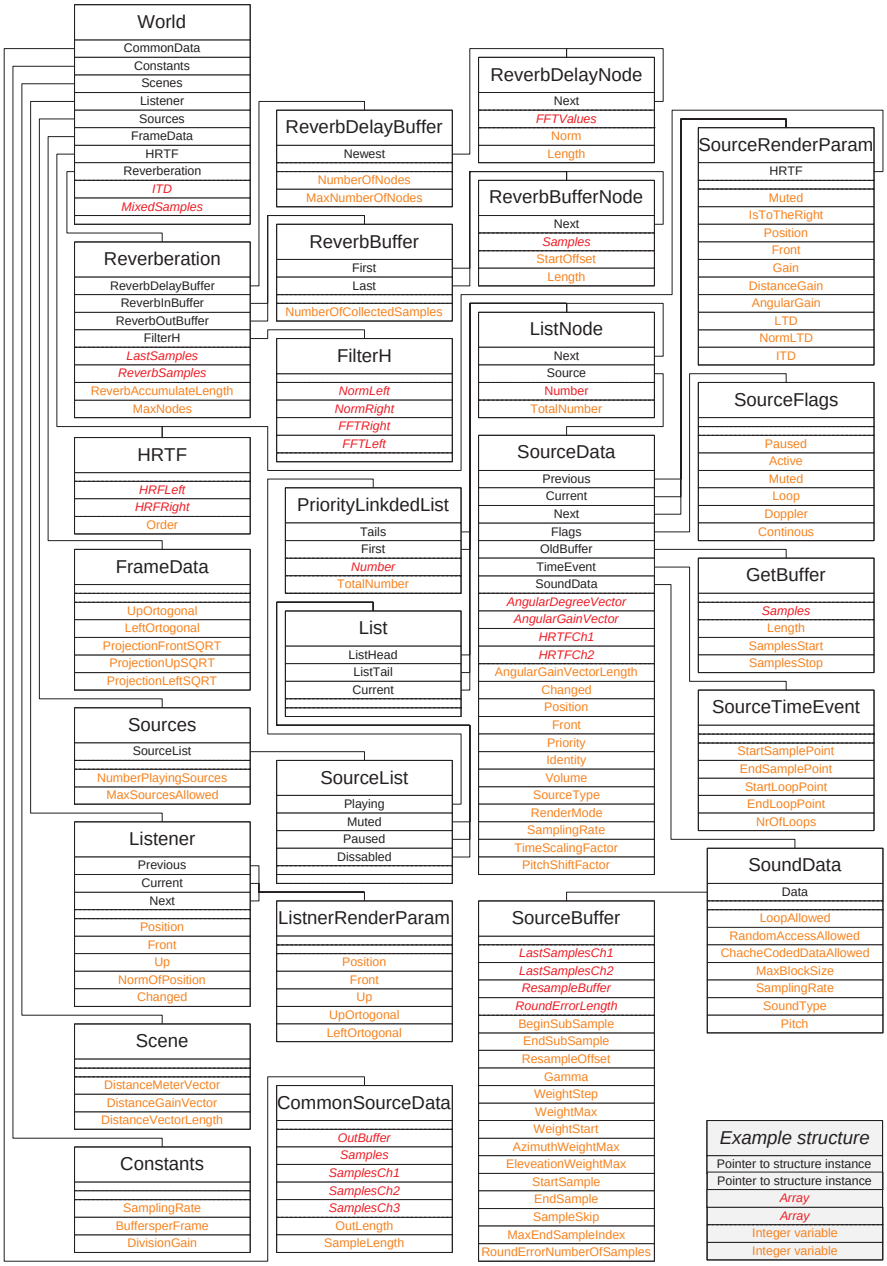

Figure 3.9: Overall structure diagram. Each line reaching between structures symbolizes a unique instance. In the lower right corner, an example structure, showing the three different parts of a structure, each separated by a doted line.

Finally the output buffers from all sources and the out-buffer from the Reverberation block are mixed into two channels (left and right channel).

3.1.4.2 Buffers in the Buffer Update

To process input samples from sources, the buffer update function needs to collect samples from a source, to perform resample, Head Related Filter (HRF), reverberation and to mix the output. This means that the buffer update needs five temporary buffers. Two of these buffers are common for all sources:

- mixedSamples of length $2 \cdot L_o$ to store mixed samples.
- tmpOut of length $2 \cdot L_o$ to store reverberation.

And the other three are not common:

- samples of length $L_b + L_m + 1$ to collect in-data from getBuffer.
- samples-ch1,2,3 each of length L_o to handle result from resample.
- outbuffer of length $2 \cdot L_o$ to handle result from HRF transform.

3.1.4.3 HRTF

To be able to perform HRF the last $L_h - 1$ samples each round is saved, and used next round.

3.1.4.4 Resample

Since the ITD causes a sample offset between the channels, ITD length of samples most be saved, and in order to perform Sinc interpolation, there is also a need to further save $Lm + 1$ samples.

3.1.4.5 Reverberation

In order to get the correct input length, input data to the reverberation is accumulated for all sources and the block is buffered. In order to get the correct output length, out-data blocks are buffered and larger or smaller chunks can be read. Each FFT transformation of a chunk of input block gives $2 \cdot L_f + 2$ FFT samples. An IFFT transformations of the $2 \cdot L_f + 2$ will result in L_f output samples. Due to that L_f is rather small, N_d output chunks from FFT transformations are saved, in order to get the reverberation delay effect, giving the delay time $\frac{L_f \cdot N_d}{S_r}$ seconds.

3.2 Implementation Parts

This section describes how the different parts of the 3D Audio Engine are implemented. A part of the A3D Execute routine is explained, followed by a implementation description in both Frame Update and Buffer Update routines. Before reading this section, readers are recommended to read and understand the entire flow of A3D Execute i.e. to read section 3.1.

3.2.1 Lists

In order to handle different sound source states without extensive use of flags and checking of flags. The sound sources will be inserted into different lists, depending on their status, see figure 3.11. When a sound source has been created and is introduced to the world it is inserted into the disabled sources list. All lists consists of the same type of nodes, this means that it is cheap to move a source from one list to another.

3.2.1.1 Playing Sources List

A sound source placed within the playing sources lists will be updated by the renderer and the samples from the source will be collected, parsed and outputted. If a sound source when updated is found to be out of range, the source will be moved to the muted sources list. The list is organized by priorities, there are 5 different levels of priorities, this means that when the list is parsed by the renderer and the CPU time available is low, there is a possibility to stop collecting and parsing samples at a specific priority, though all sources will still be updated.
The playing sources list is created as a simple linked list, priorities are achieved by having priority pointers point to the last source of each priority. When a source is inserted, it will be inserted last in the corresponding priority part of the list. When the list shall be parsed by e.g. the render, it is done in the same matter as a normal simple linked list, though there are possibilities to check for the end of a specified priority. If no source of a specified priority exists, the priority pointer will be NULL.

3.2.1.2 Muted Sources List

A sound source placed within the muted sources lists will be updated by the renderer and the samples from the source will be skipped. The sources within the muted list, will be checked for changes that should move the source to the playing list.

3.2.1.3 Paused Sources List

Sources in the paused list will not be updated by the render, to move the source back to the playing list an API call most be made. The sources in the paused list will retain the information last calculated by the render.

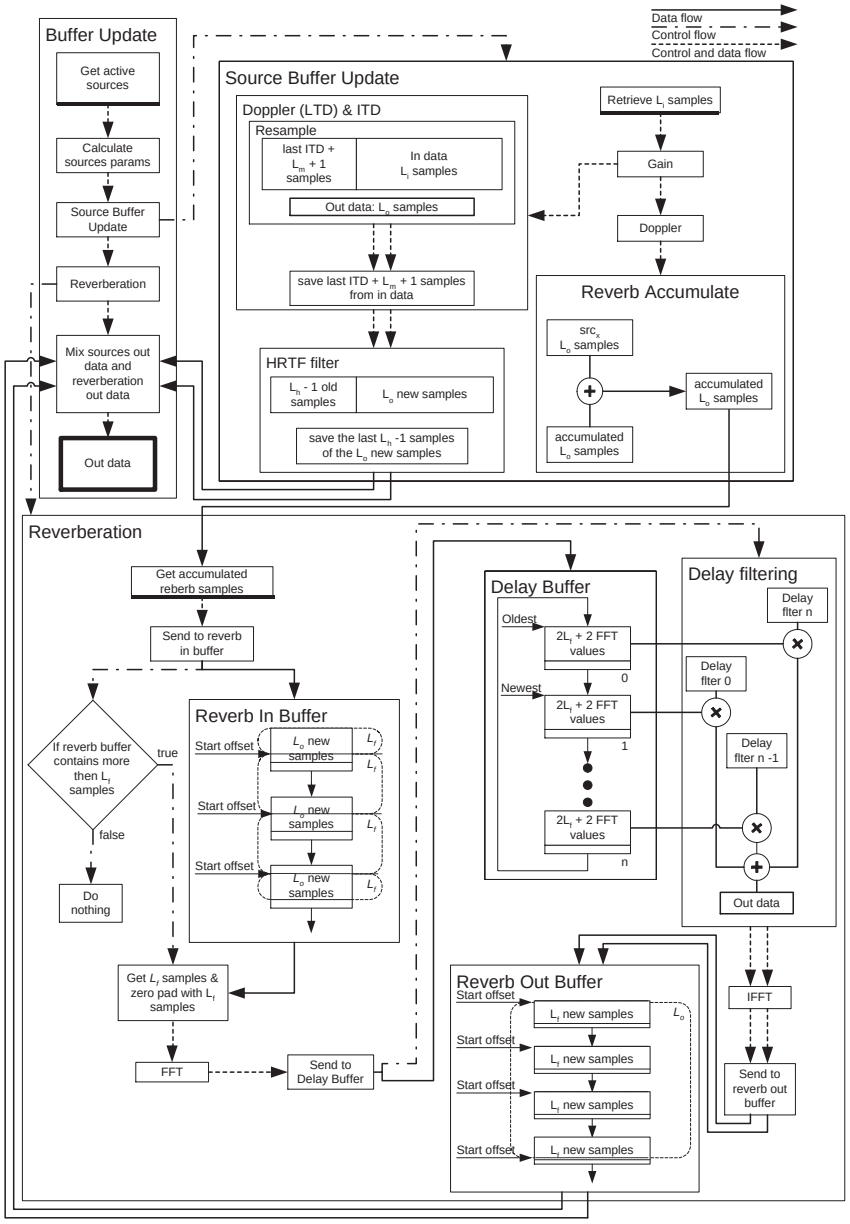

Figure 3.10: There are three major blocks in this figure: Buffer Update, Source Buffer Update and Reverberation. Buffer Update is the main flow and calls both Source Buffer Update and Reverberation. Source Buffer Update includes all calculations that modifies the samples of a source i.e. ITD, Doppler, Gain and HRTF. Samples from all sources are modified by Doppler and Gain and then accumulated into a separate reverberation input channel. After applying the Reverberation, all sources- and reverberation channels are mixed together into two channels.

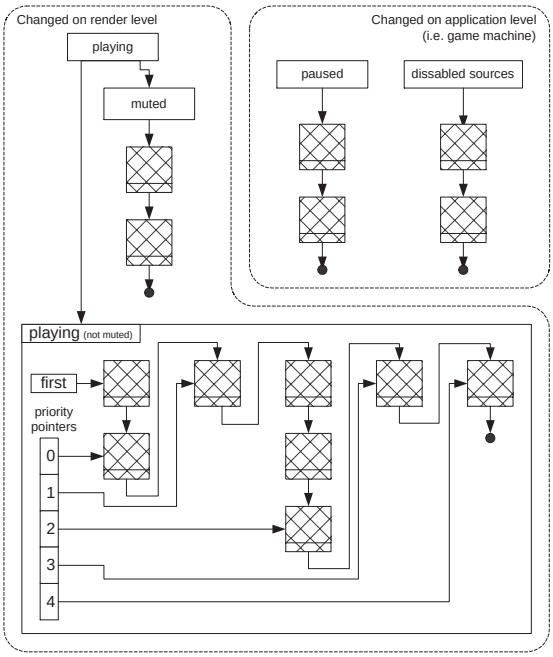

Figure 3.11: Showing usage of the source list structures. In the playing list it is possible to see the intended use of the priority pointers. They are used to point at the last source of a specific priority.

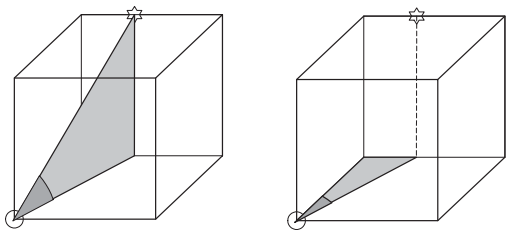

Figure 3.12: Elevation and azimuth angles. The elevation angle gives a listener information about the vertical position of a sound source. The listener experiences the sound coming from above, in front or below him. The azimuth angle gives a listener information about the horizontal position of a sound source. The listener experiences the sound coming on any direction of the horizontal plane i.e. 360 degrees around him. The circle represents a listener listening to a sound source (the star).

3.2.1.4 Disabled Sources List

A sound source placed in the disabled sources list will be totally disabled, i.e. no updates in the render will effect the source. Meaning that a source will when activated still be placed at the same position. To move the source to the muted list an API call most be done. The sources in the disabled list will have a minimum amount of memory allocated, i.e. only the most significant information will be stored.

3.2.2 Geometrical Calculations

The 3D Audio geometrical calculations are similar to those in 3D-graphics, since both sets of calculations take place in a 3D-world. Instead of one person/camera seeing graphics objects, 3D Audio has one *listener*, listening for sound waves. In 3D Audio all sound waves are produced by sound sources. Depending on the positions of the sound sources a listener may experience sounds coming from different directions. The angle calculations result in two angles: *elevation angle* and *azimuth angle*, see figure 3.12. When the listener is standing straight up, the elevation angle reveals a sound source's position on the vertical plane and the azimuth angle the sound source's position in the horizontal plane. Together, these two angles specifies the direction of arrival of the sound waves.

There are several complex calculations to be made in order to calculate elevation and azimuth angles. These calculations include aspects of vector arithmetics and are done in the Frame Update routine, i.e. no angle calculations are needed in the Buffer Update. When a sound source is created via the API, several parameters are set. Some of the parameters are used to orient the sound source. For instance, a front vector is needed since a sound source may produce sound in selected directions. The listener is a more complex matter. Apart from a front vector, a listener must have an up vector enabling the listener to tilt its head. In other words, the front vector direction represents where a person's nose is pointing at and the up vector represents the direction out from a person's crown. Please refer to figure 3.13 for an illustration.

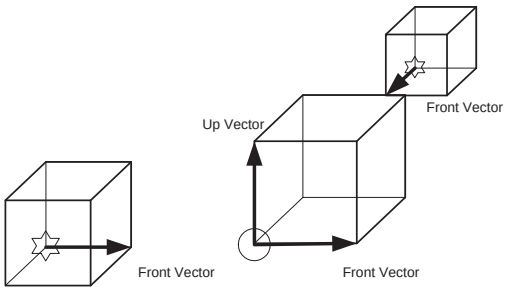

Figure 3.13: Orientation with vectors. This figure illustrates two sources (stars) and one listener (circle). The first source is behind the listener producing sound right into the listeners neck. The other sound source is to the listeners left, producing sound right into the listeners left ear.

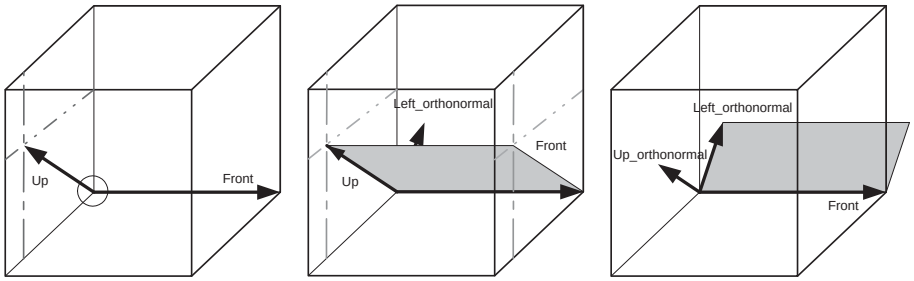

Figure 3.14: Orthonormalizing vectors. The cross product of two vectors produces a third vector which is perpendicular to the first two vectors. This is utilized to create the vectors *left orthogonal* and *up orthogonal*. The figure to the left illustrates the three positioning vectors up, left and front. The figure in the middle shows *left orthogonal* and the figure to the right shows *up orthogonal*.

Unfortunately, there is no guarantee that the listener's up and front vectors are perpendicular to each other. If the vectors are orthonormal, angles between a source and the listener may be calculated by using simple trigonometry. The front vector functions as a starting point, since it is calculated by the 3D Audio engine from several positioning inputs from the API. The cross product *up x front*, defines a new vector *left orthogonal* orthogonal to both front and up vectors. The cross product of vectors *front* and *left orthogonal*, results in the vector *up orthogonal* which is orthogonal to both front and left orthogonal vectors. The cross product calculations are shown bellow and illustrated in figure 3.14.

$$LeftOrtogonal = Listener{\rightarrow}Up \times Listener{\rightarrow}Front$$
$$UpOrtogonal = Listener{\rightarrow}Front \times LeftOrtogonal$$

When all three vectors are orthogonal, projections of the listener are calculated to acquire the correct perception of the audio scene. This is necessary since the listener may be upside down, rotating or perhaps tilting his head.

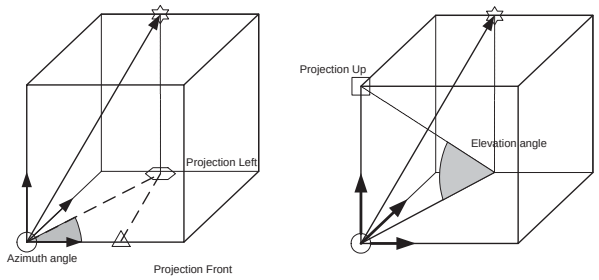

Figure 3.15: Projection of listener. The figure shows the different projections of the listener. These projections facilitates the geometrical calculations. The angles elevation and azimuth are calculated by forming right angled triangles with the listener as starting point and projections at each side. Once both sides of a triangle are know, the angle is calculated using the trigonometric function atan.

The projection points are calculated by using the scalar product between two vectors and normalizing the calculated point with one of the used vectors. The listener's projections are shown below and illustrated in figure 3.15.

$$
\begin{aligned}
ProjectionFront &= \frac{DistanceToSource \cdot Listener{\rightarrow}Front}{||DistanceToSource \cdot Listener{\rightarrow}Front||} \\
ProjectionUp &= \frac{DistanceToSource \cdot UpOrtogonal}{||DistanceToSource \cdot UpOrtogonal||} \\
ProjectionLeft &= \frac{DistanceToSource \cdot LeftOrtogonal}{||DistanceToSource \cdot LeftOrtogonal||}
\end{aligned}
$$

DistanceToSource is the relative distance from the listener to the source i.e.:

$$DistanceToSource = SoundSource{\rightarrow}Position - Listener{\rightarrow}Position$$

The calculated projections are used to compute the azimuth and elevation angles to a sound source. Note that the projections front, up and left are scalar values and not vectors. Projection left and Projection front are used to form a right angled triangle (refer to figure 3.15) and makes it possible to calculate the azimuth angle by using the trigonometrical function arcus tangent. Likewise, projection up and projection left are also used to form a right angle triangle that gives the elevation angle. The azimuth and elevation angles are computed using the following code:

$$
\begin{aligned}
NormHorizontal &= \sqrt{ProjectionFront^2 + ProjectionLeft^2} \\
AzimuthAngle &= \arctan2(ProjectionLeft, ProjectionFront) \\
ElevationAngle &= \arctan2(ProjectionUp, NormHorizontal)
\end{aligned}
$$

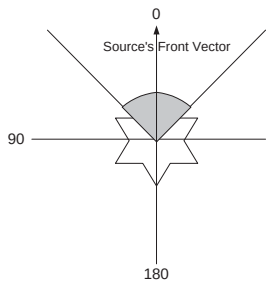

Figure 3.16: Angular gain. The angular gain of this source is set to 0 to 45 degrees. This source produces sound 90 degrees in the front direction. Listeners outside this range will not hear sound coming from this source.

NormHorizontal is the distance from the listener to the ProjectionLeft (refer to figure 3.15).

The azimuth angle may result in a negative value i.e. a value of -20 instead of 340 degrees. If the azimuth angle is less than zero, 360 degrees are added, since the 3D Audio engine only handles positive azimuth angles. The elevation angle may vary from -90 to 90 degrees, where 0 degrees means there is no elevation. The supported ranges of the azimuth and elevation angles are defined by the HRTF-filter database.

3.2.3 Angular Gain

Angular gain is used to control the spreading of a source's sound waves. Implementing this kind of gain allows a sound source to send sound waves from selected angles. The angular gain allows modification of a source's output in the range of 0 to 180 degrees, symmetrically around the source's front vector, see figure 3.16. Unlike the listener a source does not have an up vector which makes the angular gain symmetrical. For instance, if a source is modified to send sound only between 0 and 10 degrees, it will also produce sound from 360 to 350 degrees. The calculations are divided in the two different updating routines frame update and buffer update. An exact calculation of the angular gain is done in the frame update, while a linear approximation of the Next and Previous frame update values is done in the buffer update.

3.2.3.1 Frame Update Calculations

The frame update calculations uses two vectors called *angular array* and *angular gain array*. These vectors describe the spreading characteristics of a source. Table 3.2 illustrates how these vectors may look like.

The frame update calculations consists in comparing the angle between the listener and a sound source with the values in the angular vector. For instance, if the listener-source angle is 90 degrees seen from the source's front vector, using the vectors shown in table 3.2 would give the distance gain of 0.5. As in the distance gain calculations, the angular vectors specify only marginal values and linear interpolation is used to remedy the problem. For example, if the

Table 3.2: Angular- and angular gain- arrays. The table shows how a angular array (left column) and angular gain array (right column) may look like. With these set of values, the listener would perceive the sound coming from this sound source as weaker, as the angle from the source's front vector to the listener increases. The angular vector must always be increasing with a maximum value of 180 degrees. The angular gain array must always be decreasing, but does not have to reach zero.

Angle to listener	Gain
45	1.0
90	0.5
135	0.25
180	0.0

listener-source angle is 67 degrees, using table 3.2, the two closest values would be 45 degrees with a gain of 1.0 and 90 degrees with a gain of 0.5. A linear interpolation of these values would give a gain of about 0.75. The calculation used for interpolation is:

$$AngularGain = \frac{(AngleToListener - angle1) \cdot gain2 + (angle2 - AngleToListener) \cdot gain1}{angle2 - angle1}$$

AngleToListener is the angle, in degrees, between the listener and a source. *Angle1* and *angle2* are the closest values in the angle vector so that:
$angle1 < AngleToListener <= angle2$, where angle1 is as large as possible and angle2 is as small as possible.
If the angular gain vector does not reach zero, the last defined gain is used up to 180 degrees.

3.2.3.2 Buffer Update Calculations

As in the distance gain calculations, the only angular gain calculation in the buffer update is a linear interpolation between the previous angular gain and the next angular gain. Since there are a known number of steps between Previous and Next data calculations, it is very simple to calculate the approximated current angular gain. The linear interpolation between the calculated Next- and Previous- frames is described in section 3.1.2.

3.2.4 Distance and Distance gain

Distance gain is the variation of the sound volume of a source depending on the distance between a source and the listener. The distance gain calculation allow different environments to absorb different amount of sound energy depending on the distance between a sound source and a listener. A listener experiences a sound weaker the farther the sound is from him. A sound wave that travels through air looses energy to its surroundings, decreasing the sound wave's amplitude. Depending on, for instance weather conditions or environmental

characteristics, the same sound at the same distance may seem louder or weaker to the same listener.

The distance gain calculations are divided in the two different updating routines frame update and buffer update. An exact calculation of the distance gain is done in the frame update, while a linear approximation is done in the buffer update.

3.2.4.1 Frame update calculations

The distance gain is based on the the distance between a source and the listener. The distance between a source and the listener is a bit problematic to calculate. The bit depth of the position vectors of all 3D objects, is 32-bit. To calculate a distance in 3D space between two objects the following calculations may be used:

$$distance \quad = \quad ||obj1 - obj2||$$

All position vectors are 32-bit, i.e. $(obj1- > x - obj2- > x)^2$ requires 62-bits. When all values are added together, and before square root, the amount of bits required are 64-bits. Finally, the square root function returns a 32-bit value. The problem with this calculation is the maximum bits required: 64-bits. The DSP (C55x) has a 40-bits accumulator and cannot handle 64 bits. The result would be saturation to 40 bits. Since the target platform for this thesis is a C55x, this calculation was not feasible. The solution to this problem was to keep every term $(obj1- > x - obj2- > x)^2$ as 30-bits. This was accomplished by dividing the calculation into different steps. First step was to calculate $(obj1- > x - obj2- > x)$, $(obj1- > y - obj2- > y)$ and $(obj1- > z - obj2- > z)$ and storing the value as a 16-bit mantissa and a norm, i.e. in the form $mantissa \cdot 2^{norm}$. The mantissa is saved into a 16-bit vector and the smallest norm was set to all elements in the vector, giving the best possible accuracy to all three elements.

```
//(obj1->z - obj2->z) is stored in the vector.
TempVector32[2] = Source->Pos[2] - Listener->Pos[2];

//TempVar32 is the largest element in the vector.
TempVar32 = TempVector32[2];

for(k = 1; k >= 0; k--)
{
  //The values are stored into the vector.
  TempVector32[k] = Source->Pos[k] - Listener->Pos[k];

  //Store the largest element in the vector.
  if(abs(TempVar32) < abs(TempVector32[k]))
    TempVar32 = TempVector32[k];
}

//The largest element gives the smallest norm.
NormOfListenerSourceVector = norm_l(TempVar32);

//Store the mantissa values into a 16-bit vector.
for(k = 2; k >= 0; k--)
  ListenerSourceVector[k] = L_shl(TempVector32[k], NormOfListenerSourceVector - 16);
```

The 16-bit mantissa values are stored in the ListenerSourceVector and the norm is stored into NormOfListenerSourceVector. The next step is to calculate

Table 3.3: Distance- and distance gain- vector. The table shows how a distance vector(left column) and distance gain vector(right column) may look like. In this particular 3D Audio scene, the amplitude of a sound wave starts decreasing after 5 meters. When the distance between the listener and a sound source is 100 meters, the amplitude of a sound wave coming from that source would be only 1/4 of its original value. If the listener-source distance is greater or equal to 800 meters, the sound waves of that sound source will not reach the listener.

Distance in meters	Gain
5	1.0
50	0.5
100	0.25
200	0.1
800	0

$(ListenerSourceVector[0])^2+(ListenerSourceVector[1])^2+(ListenerSourceVector[2])^2$. Since the ListenerSourceVector is 16-bit, the bit depth of this calculation is 34-bit. The solution to the problem is presented below:

```
//Accumulate values before square root, keeping TempVar32 as 32-bits.
TempVar32 =  (ListenerSourceVector[0]*ListenerSourceVector[0])>>2;
TempVar32 += (ListenerSourceVector[1]*ListenerSourceVector[1])>>2;
TempVar32 += (ListenerSourceVector[2]*ListenerSourceVector[2])>>2;
```

By shifting down the product of the vector by 2 steps, the two least significant bytes are truncated and a 30-bit value is achieved. The resulting Tempvar32 variable will contain a 32-bit value at the end of the code. The ListenerSourceVector contains mantissa values with the same norm. To preserve the correct value in the TempVar32 at the end of the code, a norm for TempVar32 must be created.
The final step was to calculate the square root. The square root function gets a 32-bit value and return a 16-bit value. This procedure guarantees that the bit depth never exceeds 32-bit.
To be certain that the precision was sufficient, a test was made comparing the distance in this implementation to a floating point calculated distance. The test consisted in an object moving along an oval trajectory, 1000 meters wide and 20 meters in height. The distance values were saved to file and plotted in Matlab. The maximum difference was 0.2 meters at the distance of 500 meters.

Distance gain calculations uses two vectors called: distance vector and distance gain vector. These vectors describe the environmental characteristics in a world/room and are connected to a specific 3D Audio scene. When the 3D Audio engine changes a scene, new distance- and distance gain- vectors are loaded. Table 3.3 illustrates how these vectors may look like.

The frame update calculations are similar to the angular gain calculations. The angular gain calculations are described in section 3.2.3.

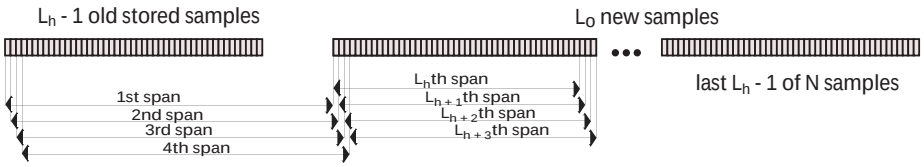

Figure 3.17: Applying HRTF FIR filter. Figure shows how the filter is applied and how the samples from last run is used.

3.2.4.2 Buffer update calculations

The only distance gain calculation in the buffer update is a linear interpolation between the previous distance gain and the next distance gain. Since there are a known number of steps between Previous and Next data calculations, it is simple to calculate the approximated current distance gain. The linear interpolation between the calculated Next- and Previous- frames is described in section 3.1.2.

3.2.5 Head Related Transfer Function (HRTF)

HRTF the technique used to make the sound appear external. The mono input signal will be filtered by a filter for left ear and a filter for right ear, creating a stereo output. The filter values have been stored in such a way that when calculating the absolute sum for each of the HRTF-filters, i.e. all L_h filter taps summarized, the sum is always possible to store with Q.15. Since the HRTFs are saved as transfer functions they will be implemented as FIR filters of length L_h. In order to filter a sample X, $L_h - 1$ previous samples of X must exist, yielding that all the time there is a need to keep track of $L_h - 1$ previous samples per channel (left and right) $2 \cdot (L_h - 1)$ samples per source. See figure 3.17.

3.2.5.1 HRTF Filter Loop

The filter is placed with start on the $X - (L_h - 1)$:th sample and with end on the X:th sample. Each sample between the $X - (L_h - 1)$:th sample and X:th sample are multiplied with the filter tap value corresponding to the position in the filter. And these values are accumulated and will be the new sample value for X. See figure 3.17.

$$
\begin{aligned}
Ch_1\_sample_i &= \sum_{k=0}^{L_h} hrfilterLeft[i-k] \cdot samples_ch1[k] \\
Ch_2\_sample_i &= \sum_{k=0}^{L_h} hrfilterRight[i-k] \cdot samples_ch2[k]
\end{aligned}
$$

3.2.5.2 Frame Update Calculations

The correct HRTF set to use is calculated through 4-point interpolation.
if a value is needed for a point, x, and the value for that point is not in the discrete set, it is possible to use those points surrounding x in order to get

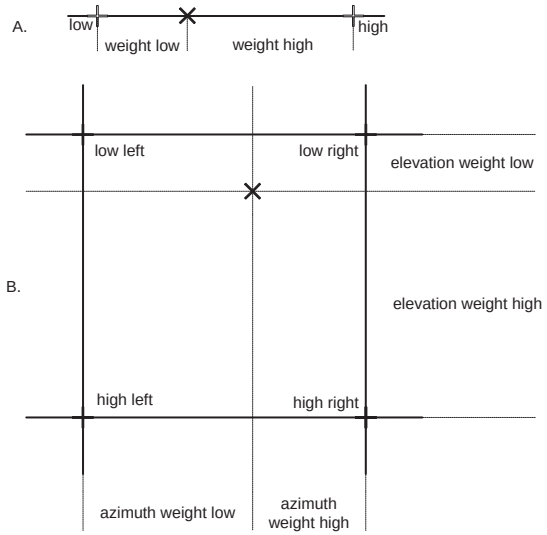

Figure 3.18: This figure illustrates two kind of interpolation used in this thesis. A. 2-point interpolation, B. 4-point interpolation.

an approximation of the value. This is called interpolation, two basic types of interpolation are used; 2-point interpolation (where the unknown point is in the same line as the points surrounding) and 4-point interpolation (where the unknown point is in the same plane as the surrounding points), see figure 3.18. The four nearest discrete points of x are found, (upper left, upper right, lower left, lower right) and are weighted together. The weight used is the fraction part of the point x. The value for x is calculated by 2-point interpolation first of the horizontal values (i.e. upper left to upper right and lower left to lower right), this gives two vertical values that are calculated by 2-point interpolation (i.e. the upper value is weighted with the lower value) giving the interpolated value.
Each time a new frame update is done, the previous stop HRTF will act as the start HRTF.

3.2.5.3 Buffer Update Calculations

This is done by 2-point interpolation, between the start HRTF and the stop HRTF, weighted depending on the number of buffers processed since last frame update. The two nearest discrete points of x are found (i.e. upper and lower or left and right), these values are weighted together using the fraction part of the x as weight (the weight used can also be a distance from a point and its complement). The fraction part of x is denoted f, Two discrete points surrounding x; p_1 and p_2, The value of p is denoted $v(p)$.

$$v(x) = v(p_1) \cdot f + v(p_2) \cdot (1 - f). \tag{3.1}$$

FIR filter frequency response is determined by a set of coefficients (filter taps). The purpose of the coefficients is to alter the signal content by means of simple arithmetic. In 3D Audio, the Head Related Transfer Function(HRTF)

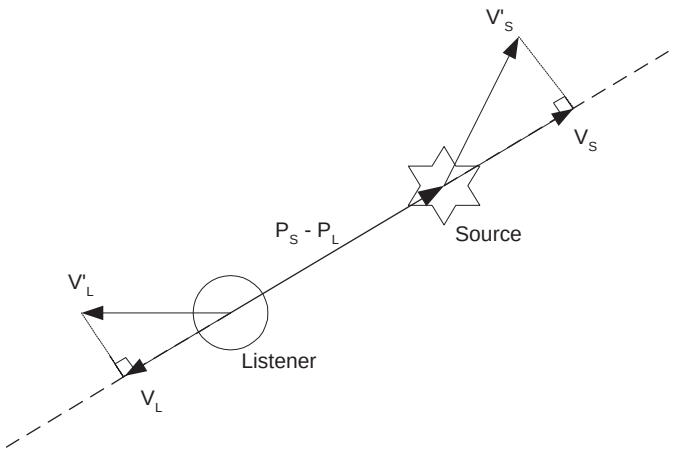

Figure 3.19: The velocities of the listener(V_L') and a source(V_S') are projected as V_L and V_S to the same plane as the calculated distance vector.

uses a set of filter taps to transform incoming samples so that they are experienced as coming from a certain angle or direction.

3.2.6 Locational Time Delay (LTD) and Doppler

LTD is used by the 3D Audio engine to create Doppler effect. The LTD is an amount, that is added/subtracted to obtain more or lesser samples from a source into the input buffer. Despite the number of samples obtained, a resampling function resamples the input buffer into L_o samples. If more than L_o samples are present in the input buffer, they are sampled down to L_o. This will be experienced as a higher pitch. Likewise, when less than L_o samples are present in the input buffer, they are sampled up to L_o, thus giving the impression of a lower pitch. The LTD is calculated each frame and interpolated each buffer. The calculations used in this implementation use velocity vectors that are set by a user via the API. This approach uses less computational complexity than, for instance using positioning vectors to calculate velocity vectors that are used to compute the LTD value. The implementation described below is the actual implementation of the LTD calculations. Due to problems described in section 4.1.1 the first implementation of the LTD calculations was replaced by the actual one. The first implementation is described in section A.1

3.2.6.1 LTD on Frame Basis

The LTD is calculated in every frame, the value depends on the distance to the source and the speed of sound in the medium that the source and listener are located in. For each frame a new value is calculated and the older value becomes the previous value, giving next- and previous LTD values. The LTD calculations consists of projecting the velocity of a source and the listener into a distance positioning vector. The calculations are described below and illustrated in figure 3.19.

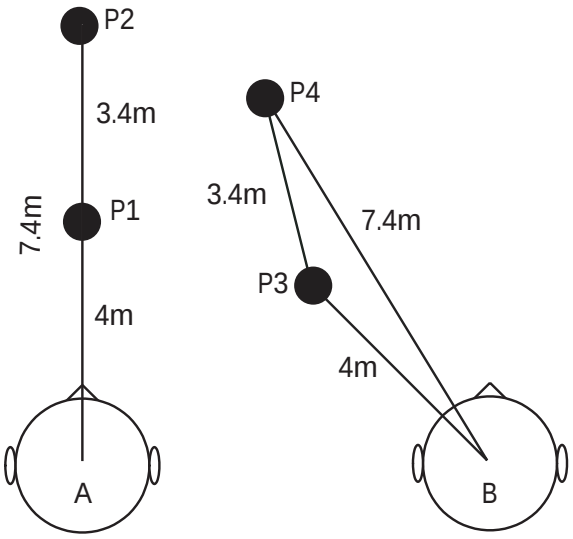

Figure 3.20: Doppler and ITD effects (maximum allowed movement), see table 3.4

The Distance calculations in section 3.2.4, compute the distance vector $P_S - P_L$, the distance between a source and the listener. The velocity vectors set via the API, V_L' and V_S', are projected on the distance vector. The resulting scalar velocities are: V_L and V_S respectively. The Doppler factor is calculated by dividing the alterations caused by V_L and V_S to the speed of sound, c, as shown below:

$$Doppler\_Factor = \frac{c - V_L}{c + V_S} \tag{3.2}$$

To use the Doppler factor in this 3D Audio implementation, it must be converted into samples. By multiplying the Doppler factor with the number of samples in the buffer, results in the number of samples needed to create the Doppler effect.

3.2.6.2 LTD on Buffer Basis

For each buffer a current value is calculated, the next and previous values are for each buffer weighted together through 2-point interpolation shown in section 3.1.2..

3.2.7 Interaural Time Difference (ITD)

The ITD is used to further, combined with the HRTF, enhance the experience of directional sound. The ITD is the time difference between signals to the ears. The ITD is implemented as a sample delay between the two channels in use.

Table 3.4: Maximum possible change in LTD and ITD with limits to speed of sound in air and using a 10ms buffer (480 samples long). Table A and B corresponds to A and B in figure 3.20. The values $LTD\Delta$ and $ITD\Delta$ shown in table A and B is the number of samples that must be retrieved in order to perform movement between point P_1 to P_2 for table A and respectively P_3 to P_4 for table B.

A			
LTD_{p_1}	509	ITD_{p_1}	0
LTD_{p_2}	988	ITD_{p_2}	0
$LTD\Delta$	-479	$ITD\Delta$	0
$LTD\Delta + ITD\Delta$		-479	

B			
LTD_{p_3}	509	ITD_{p_3}	17.5
LTD_{p_4}	988	ITD_{p_4}	8.75
$LTD\Delta$	-479	$ITD\Delta$	8.75
$LTD\Delta + ITD\Delta$		-470.5	

3.2.7.1 ITD on Frame Basis

The ITD value is calculated with 4-point interpolation, as described in 3.2.5.2. To mark that the ITD is on the left hand of a listener the ITD value is multiplied by -1, thus the value is negative on the left hand of a listener and positive on the right hand. The ITD value is stored in Q-16 in order to get good resolution. Each time a new frame update is done, the previous stop ITD will act as the start ITD.

3.2.7.2 ITD on Buffer Basis

The current ITD is calculated through 2-point interpolation, between the start ITD and the stop ITD, weighted depending on the number of buffers processed.

3.2.7.3 ITDΔ

ITDΔ is the difference between the last applied ITD and the new calculated ITD, i.e. ITDΔ only exists in calculations applied for each buffer. The value of ITDΔ will effect the number of samples that are needed. When a sound source moves toward a point in the dead center in front of the listener, the ITD will decrease. This will cause the ITDΔ to get negative, thus making the delay between left and right channel smaller. Hence fewer samples than in the previous buffer are needed. The opposite, when a sound source moves away from a point in the dead center in front of the listener, the ITD will increase. This will cause the ITDΔ to get positive, thus making the delay between left and right channel larger. More samples than in the previous buffer are needed. If the sound source is placed in the dead center in front of the listener, the sample delay between left and right channel is zero, thus both ITDΔs are zero. The ITDΔ behavior is illustrated in figure 3.21.

Depending on the value of ITD for the last frame (ITD_0), the value is stored in either ITDΔ_0 or ITDΔ_1. The ITD for the current frame is calculated and is, depending of the value (ITD_1), either subtracted or added to $ITD\Delta_0$ respectively $ITD\Delta_1$, shown in table 3.5.

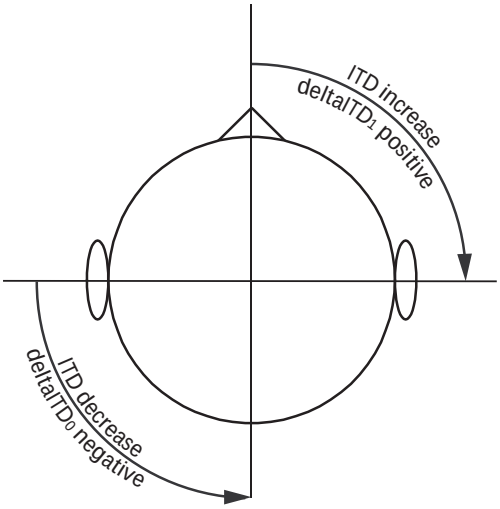

Figure 3.21: ITD movement. Arrows show how ITD and ITDΔ are affected by circular movement of a source. When a source moves from a point right in front of a listener towards one of the sides, the ITD will increase and the corresponding ITDΔ for that side will be positive. When a source moves from one of the sides towards a point in front or behind of the listener the ITD will decrease and the corresponding ITDΔ will be negative.

Table 3.5: ITD calculations and the effects on $ITD\Delta$. When comparing the last ITD and the new ITD, the effects on ITDΔ_0 and ITDΔ_1 will be different depending on the signs of the last ITD and the new ITD. The sign of ITD shows on which side of the head the source is located. Assignment will be denoted $:=$ and simplification will be denoted $=$.

		After	
		$ITD_1 < 0$	$ITD_1 > 0$
Prior	$ITD_0 < 0$ $ITD\Delta_0 := 0$ $ITD\Delta_1 := -ITD_0$	$ITD\Delta_0 :=$ $-ITD\Delta_0 = 0$ $ITD\Delta_1 :=$ $ITD_1 - ITD\Delta_1 =$ $ITD_1 + ITD_0$	$ITD\Delta_0 :=$ $ITD_1 - ITD\Delta_0 = ITD_1$ $ITD\Delta_1 :=$ $-ITD\Delta_1 = ITD_0$
	$ITD_0 > 0$ $ITD\Delta_0 := ITD_0$ $ITD\Delta_1 := 0$	$ITD\Delta_0 :=$ $-ITD_0$ $ITD\Delta_1 :=$ $-ITD_1 - ITD\Delta_1 =$ $-ITD_1 - 0 = -ITD_1$	$ITD\Delta_0 :=$ $ITD_1 - ITD\Delta_0 =$ $ITD_1 - ITD_0$ $ITD\Delta_1 :=$ $-ITD\Delta_1 = 0$

Table 3.6: Maximum step per buffer update in air. Due to the limit of speed of sound in the resample function the total step size within one buffer update is limited. The table shows the limit depending on the size of the out buffer used.

Buffer time	*Buffer size*	*Maximum step per buffer*
5 ms	240 samples	1.7 m
10 ms	480 samples	3.4 m
20 ms	960 samples	6.8 m
50 ms	2400 samples	17.0 m
100 ms	4800 samples	34.0 m

3.2.8 Input Buffer Handling

Due to LTD (Doppler) the maximum step to take within a buffer depends on the buffer size chosen and the medium (the speed of sound is not equal in different mediums) the listener and sound source are in. Normally the medium used is air, maximum step size is shown in table 3.6.

At the beginning of each buffer update the length is calculated and the end sample position is updated for each channel. Calculations on number of input samples and end position is performed in subsamples and depends on the buffer size, the ITDΔ and the LTDΔ, hence the number of samples can differ between right and left channel.

```
Length[ch] = DeltaITD[ch] + DeltaLTD + buffersize;
endSubSample[ch] = beginSubSample[ch] + Length[ch];
```

At the end of each buffer update the begin sample position is updated, this is performed by subtracting the least of the two channels end positions from both channels end positions.

```
beginSubSample[ch] = endSubSample[ch] -
                     min(endSubSample[ch1], endSubSample[ch2]);
```

3.2.9 Sinc Interpolation

Since the output buffer size, L_o, is constant and input data size, L_i, may vary there is a need to make the L_i sized input of size L_o, this is done by Sinc interpolation.

Basically the technique selects a set of size L_o in-data and applies a Sinc shaped filter to this set. There are two cases for interpolation; the in-data length may be smaller than the out-data length and the in-data length may be larger than the out-data length. The first case is denoted upsampling and the second case downsampling.

This Sinc interpolation is done with a Sinc length L_m (e.g. $L_m = 6$), i.e. for each sample parsed $2 \cdot L_m + 1$ sample calculations will be performed. The γ value symbolized the in/out relation, it is used to determine whether upsampling or a downsampling shall be applied.

$$\gamma = L_i / L_o. \tag{3.3}$$

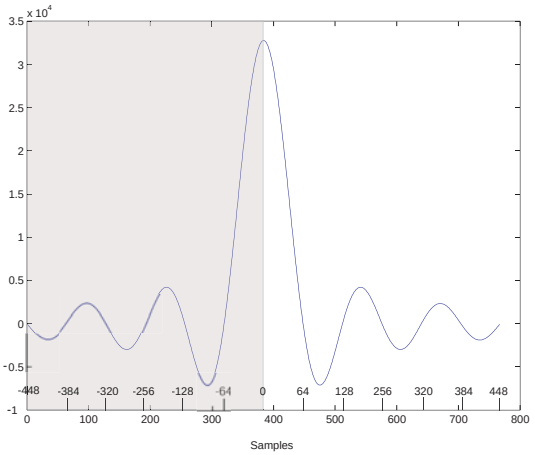

Figure 3.22: Sinc function. The Sinc used for resample is symmetric, hence only half of the Sinc needs to be stored. The non-grayed area shows the part of the Sinc that will be used.The Sinc table is oversampled by 2^{L_m}, which gives that 2^{L_m}, 2^{L_m+1}, 2^{L_m+1}, ..., corresponds to 1, 2, 3, ..., decimal.

If γ is less or equal to 1, upsampling will be applied else downsampling will be applied. Included in the resample function is also a low pass filter, in order to avoid aliasing. The Sinc tables are generated in Matlab and transformed into fixed-point values. The Sinc is created with 2^{L_m} fraction steps. Since the Sinc is symmetric it is possible to just store the second half of the Sinc in order to save space, see figure 3.22.

3.2.9.1 Index and Fraction

The in-data will be parsed with an index that is updated for each output sample. The index has a integer part and a fraction part. The integer part of the index is a pointer to a sample in the input buffer. The Sinc filter will be applied to this sample and thereafter to L_m - 1 previous in-data samples and to L_m - 1 (forthcoming) in-data samples. The Sinc filter will initially be indexed with the fraction part, the middle sample will be multiplied with the initial Sinc value. For each of the Lm - 1 previous and L_m - 1 (newer) in-data samples, the index into the Sinc table will be increased or decreased by γ^{-1} before applied.

The in-data sample selected by index will simply be multiplied by the Sinc value selected by fraction. For those $2 \cdot (L_m - 1)$ samples surrounding sample index, the values will be multiplied by the corresponding Sinc values, see equation 3.4. Since just half the Sinc is stored, the fraction parts for the previous samples and the newer samples will differ. This means that it is necessary to transform the fraction index that should have been used on the negative side of the Sinc to a positive value, once this transformation is done, the negative fraction index can also be increased for each step. The fraction part used to index the positive side of the Sinc is called $fraction_u$ for $fraction_{upper}$ and the part used to index the negative side is called $fraction_l$ for $fraction_{lower}$. Initially $fraction_u = fraction$ and $fraction_l = \gamma^{-1} - fraction$.

$$
\begin{aligned}
samples_{out}[m] \quad = \quad & samples_{in}[index] \cdot sinc[fraction] + \\
& \sum_{k=1}^{L_m} (samples_{in}[index - k] \cdot sinc[fraction_u + k \cdot \gamma^{-1}] + \\
& \qquad samples_{in}[index + k] \cdot sinc[fraction_l + k \cdot \gamma^{-1}]) \qquad (3.4)
\end{aligned}
$$

3.2.9.2 Limits

There are both upper and lower limitations of the resample function. The upper limit is near twice the sample rate, i.e. near twice the speed of sound. This limitations are due to the algorithm implemented, since the algorithm has no description of what to do when more then twice or less then half resampling is demanded. At the higher limit the actual arrival of samples to the listener would be inverse. At the lower limit, the need to repeat the same sample over and over again, would cause the quality to be very low.

3.2.10 Reverberation

Implemented is medium long reverberation of type frequency domain filtering using Overlap and Save. The reverberation is implemented to be $N_d \cdot L_f$ samples long per channel i.e.

$$\frac{N_d \cdot L_f}{S_r} \, seconds \qquad (3.5)$$

The reverberation is performed via multiplication of samples and filter in the frequency domain, the result is then summarized and after all samples is parsed the result is transformed to time domain, see reverberation in figure 3.10.
Since the reverberation works with 2^x length and the render can work with any kind of length, there is a need to buffer input to the reverberation function, likewise there is a need to buffer output from the reverberation function, as shown in figure 3.23.

3.2.10.1 Buffers

Each source will send L_o samples to the reverberation accumulate function. After all playing sources has been parsed, the reverberation apply function will be called.

Input buffer is managed as a simple linked list, where each node holds a start offset, see figure 3.23. At the beginning of the reverberation apply function, the accumulated samples will be sent to the input buffer. The samples will be stored in a new node, with start offset set to zero. For each round of filtering, L_f samples will be retrieved from the buffer. If a retrieve call do not uses all samples within a node. The start offset will be set to the first unused sample. Once all samples in a node has been retrieved the node is deleted. Output buffer is managed as a simple linked list, where each node holds a start offset, see figure 3.23. At the end of reverbApply, L_f out samples will be sent to this buffer. At the end of the buffer update function it could be possible to retrieve L_o samples if $L_o \leq L_f$ or there were unused out samples from previous

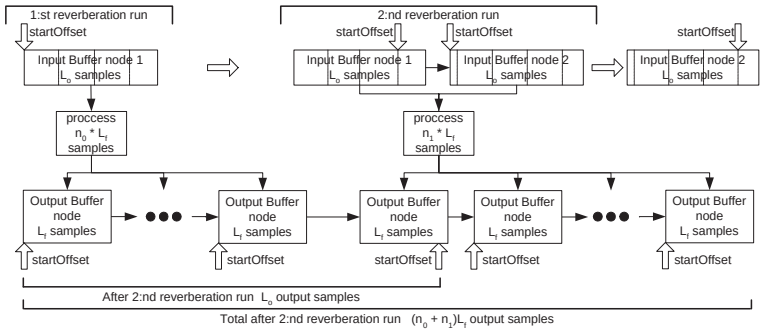

Figure 3.23: Reverberation buffer usage. Each time the reverberation is executed, as much as possible of the input data will be processed. Though there is no guarantee that the amount of data processed will be enough to create a whole output block of length *Lo*. In the case there is not enough data to produce an output block, output will be delayed until the amount of data needed is available.

execution. If not all samples in a node is used, the start offset will be changed. The delay buffer is managed as a simple circular linked list, with pointers to oldest and newest nodes, see figure 3.10. When inserting a new set of FFT values, and the list is empty, or the number of nodes is less then the maximum number of nodes, a new node, containing the FFT values will be created and the newest pointer will be updated, if there are maximum number of nodes in the list, an insert will cause the oldest node's FFT values to be deleted and the new FFT values to be stored in the oldest node, the oldest and newest pointers will thereafter be updated. Deletion of nodes in the list will only occur when all sources have stopped producing new samples.

3.2.10.2 Filters

The filters are generated from white noise and are exponentially decreasing, there are N_f filters stored per channel. The filters are unique for the left and right ear. When initializing the filters they are zero padded and thereafter FFT transformed, the FFT values and the normal is then stored.

3.2.10.3 Applying Reverberation

When applying reverberation, the accumulated samples are sent to the input buffer, and a new node for holding the samples is created. Thereafter if possible L_f samples are retrieved from the input buffer, these samples are when retrieved zero-padded with L_f samples. The $2 \cdot L_f$ samples returned will be frequency transformed through FFT, the FFT function will return $4 \cdot L_f$ values, of which $2 \cdot L_f + 2$ needs to be saved. The other half is not necessary to save since the values are complex conjugated mirror image of the lower half. These $2 \cdot L_f + 2$ values will be sent to the delay buffer.
All nodes in the delay buffer will be multiplied with the corresponding filter. The filter multiplication is divided into two parts. One part that will evaluate the normalization values (here denoted normal) of the product, to be able to get all

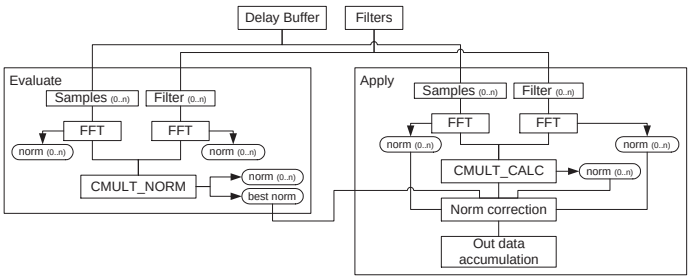

Figure 3.24: Reverberation normal calculation and compensation. To be able to perform a good normal compensation the multiplication of FFT values is performed in two steps, the first step will be used to find which normal that is best, the second step will perform the actual multiplication and there after compensate the normal.

results with the same normal. Thereafter the actual multiplication is performed and the results are normal compensated and accumulated, see figure 3.24.

The accumulated result will thereafter be transformed back to time domain through IFFT, denormalized and sent to the output buffer.See Reverberation part of figure 3.10.

3.2.10.4 Library functions

Several library functions was available. Three of these were used in the implementation of reverberation; FFT, IFFT and CMULT.

Fast Fourier Transform (FFT) and Inverse Fast Fourier Transform (IFFT), used to transform time domain samples into frequency domain samples and frequency domain samples to time domain samples. Both functions takes as arguments a vector of input samples and a vector of return samples, both functions will return a normal. Complex multiplier (CMULT), is used to multiply complex sample arrays. As arguments this function takes two input vectors and a return vector, and will return a normal. Since a number of blocks of FFT samples shall be accumulated, there is a need for the normals of these blocks to be the same. In order to select the best normal (i.e. to loose as little accuracy as possible),the normal outcome of the complex multiplication must be evaluated, the best normal must be chosen and the blocks with normal not equal to the best normal must be adjusted. When studying the code for the CMULT, it was found that the CMULT function first evaluated the normal within a block and then, after selecting the best normal actually performs the multiplication. The CMULT function could easily be split into two functions. One function that evaluates the normals, and one function that performs the multiplication. In this way it is possible to first for each block evaluate the normal, then for each block perform the multiplication and adjust the result depending on the relation to the previously calculated best normal. And then to accumulate the result.

3.2.10.4.1 Normal calculations Each of the library functions will return a normal, but not all the normals returned can be used directly to denormalize

the values. It was found when performing FFT of two simple values, multiply the values via CMULT and then perform IFFT, that the straight forward denormalization was incorrect. To be able to see a pattern, a number of different input values was used, it was then found that in order to get a correct denormalization the normal return from the IFFT needed to be compensated with -8 (compensation depending on the FFT length chosen) and the normal return from CMULT needed to be compensated with +15.

3.2.10.5 Dynamic Memory Usage

The amount of memory needed for reverberation depends on the number of delay nodes N_d, the filter length L_f, the output length L_o and the number of channels N_c. It is possible to calculate the amount of memory needed (i.e. only the buffer memory is included).

- Input Buffer - maximum amount of memory usage:

$$L_o + L_o \, mod \, L_f \tag{3.6}$$

- Output Buffer - maximum amount of memory usage:

$$\lfloor \frac{(\lceil \frac{(L_o/L_f)}{\lfloor L_o/L_f \rfloor} \rceil \cdot L_o)}{L_f} \rfloor \cdot L_f \tag{3.7}$$

- Delay Buffer - maximum amount of memory usage:

$$N_d \cdot 2 \cdot L_f \tag{3.8}$$

- Filters FFTs for the filters are stored dynamically:

$$N_c \cdot N_d \cdot 2 \cdot L_f \tag{3.9}$$

- Calculations The dynamic allocation within the reverberation apply function

$$\begin{gathered} 2 \cdot 4 \cdot L_f + 2 \cdot L_f + 2 \cdot N_d + (2 \cdot L_f + 2) + 2 \cdot (2 \cdot L_f + 2) = \\ 10 \cdot L_f + 2 \cdot N_d + 6 \cdot L_f + 6 = \\ 16 \cdot L_f + 2 \cdot N_d + 6 \end{gathered} \tag{3.10}$$

- Total maximum dynamically allocated memory

$$L_o + L_o \, mod \, L_f + L_f \cdot (\lfloor \frac{(\lceil \frac{(L_o/L_f)}{\lfloor L_o/L_f \rfloor} \rceil \cdot L_o)}{L_f} \rfloor \cdot L_f + N_d \cdot 2(1 + N_c) + 16) + 2 \cdot N_d + 6 \tag{3.11}$$

3.2.11 Approximation of acos, atan and sqrt

To keep complexity at a low level, the functions acos, atan and sqrt are linear interpolated. These functions are used in the FrameUpdate procedure to calculate angles, projections and distances. The basic algorithm is the same in all functions. The wanted function is plotted, in for instance Matlab, and a set of values that can linear interpolate the function are saved in a constant table using the Q15 format. The constant table values are used to linear interpolate the function.

3.3 Development/Testing functions

Some functions and routines used in the development phase, were created to test various functions of the 3D audio engine, and to produce sound trough the PC's sound card. These functions are not needed or must be modified to fit in the final implementation.

3.3.1 File reading/writing

The A3D_file package is a set of functions to read and write samples from/to files. The final implementation will not require any file read or write routines, since audio will be sampled in real-time from a line-in jack on the DSP platform. The output will be sent via a D/A converter to headphones.

3.3.2 Buffer Simulation

The A3D_getbuffer routine, requests samples from files. The getBuffer function will be modified to manage the incoming stream of samples from the line-in jack in the DSP platform.

3.3.3 Movement Routine

The A3D_Movement function is a test function that moves a source in a specified trajectory depending on in-parameters. In the final implementation, all position updates are done only via the API.

3.3.4 Audio device

To communicate with the sound card, an audio device called portaudio was used[7]. Port audio is an open-source audio I/O library that works on various platforms. This device was used under the developing phase on the PC. Port Audio uses DirectX to communicate with the PC's sound card to deliver sound. Port Audio was chosen at an early stage due to its simplicity and being open source. In the final implementation, no audio device is needed since the DSP will output samples via a D/A converter to a stereo jack. Headphones will be connected to the stereo jack, producing the 3D Audio experience.

3.4 API

The API is a layer between the renderer and the user application. The API consists of a collection of layer functions, intended to be used to change source, listener, environment and renderer parameters. Most API functions consists of one or more renderer function calls. This means that actions always will be done in the same effective and correct manner.

3.4.1 Design Background

The design of the API for the 3D Audio engine was inspired by the API functionality of Java 3D [8]. The Java 3D platform is well documented and some

of the API calls were found suitable. The DirectX API was also considered but Java 3D was found to have more comprehensive calls.

3.4.2 Specifications

There is a need for a number of different API functions depending on what object that shall be effected. There are four groups of API functions: renderer, environment, listener and source.

3.4.2.1 Source API Functions

API function regarding source movement and behavior. All source API function will as the first argument take a source identity.

- Move, takes as argument a new position, will at next frame update, update the next position to the new position.
- Stop, will make the source stop its movement, sound generation and place it in the disabled list.
- Start, will move the source to the playing list, start the source movement and the sound generation.
- Pause, will stop the source movement, sound generation and place it in the paused list.
- Volume, takes as argument the new volume, the volume will be set, a value of zero will make the source muted.
- Priority, takes as argument the new priority, the priority will be changed and if source is in playing list necessary movement will be done.
- Doppler, takes as argument a Doppler factor, zero will disable Doppler, one will make Doppler realistic, higher or lower value then one will emphasize or deemphasize the Doppler effect.
- Type, takes as argument the type (omnidirectional or directional)
- LimitAngular, takes two arrays as arguments, the angular array and angular gain array. this call will limit the sound direction according to the specified angels.
- FrontDir, takes as argument direction (only applied for sources of directional type), will effect in what direction sound is heard.

3.4.2.2 Listener API Functions

API function regarding listener movement and behavior.

- Move, takes as argument a new position, will at next frame update, update the next position to the new position.
- FrontDir, takes as argument direction, will effect in what direction the listener is facing.

- UpDir, takes as argument direction, will effect in what way the listener's head is tilted.
- HRFset, takes as argument a HRF set identification, will at next frame update change HRF filter set used.

3.4.2.3 Environment API Functions

API function regarding environment behavior.

- Reverberation, takes as argument length of reverberation desired, a length of zero will disable reverberation.
- DistanceGain, takes as argument a new distance gain vector and its length

3.4.2.4 Renderer API Functions

API function regarding renderer behavior.

- Volume, takes as argument the new volume, will set the overall volume.
- ActiveSources, takes as argument the number of sources allowed, will set the maximum number of allowed sources.
- CreateSource, takes as arguments parameters such as orientation vectors, Doppler, priority, Volume, etc and returns a source identification.
- DeleteSource, takes as argument a source identification, will remove the source from lists an delete the source.
- FramePeriod, takes as argument the number of buffer updates that should occur before an frame update is executed.
- Create, will create a basic render environment.
- Destroy, will stop all sources and clean up all sources, listener and environment.
- Pause, will pause all sources.

3.5 Summary

The entire 3D Audio implementation was described in this chapter. The first section described the flow of the 3D Audio renderer and why the current solution was chosen. The implementation parts, i.e. the algorithms of the various parts of the 3D Audio engine, were described in section 3.2. Other parts of the 3D Audio engine that were described in this chapter are the API and testing functions.

Chapter 4

Verified Accuracy of Implementation

This chapter compares the accuracy between the fixed point- and floating point 3D Audio implementations. Both version of 3D Audio are implemented using the same structure, functionality and flow. Matlab 6.5 was used to compare the key variables in the fixed point and floating point versions of the code. Various Matlab scripts were created in order to make the comparison between the fixed point and floating point versions of the code possible. The various Matlab scripts are described in appendix B.

4.1 Fixed point and Floating point Accuracy

To ensure the accuracy of the fixed point code, Matlab was used to create scripts of control messages to both the fixed point and the floating point code. This was used to compare key variables used in both versions of the code. The test consists in moving a sound source around a listener in a specific environment. Since both versions were coded to be as similar as possible, variables and methods could be compared relatively easy. The accuracy test was divided into four different steps. Step1, runs a Matlab script that creates a file containing initializing data, data about scene environment, source movement, listener positions, etc. In step2, the fixed- and floating- point versions are started on the PC. Both versions searches for the file created by step1 and begin initialization. This is made so that both versions have an identical scene environment. The applications continue their execution, moving around a sound source and writing various parameter values to a file. Step3, consists of a second Matlab script that loads and plots and compares the parameter values from both versions. The last step, step4, is to analyze the gathered data plotted by Matlab and to correct problems if necessary. Both frame update and buffer update were compared using this test at different developing stages. The frame update block was tested first, since the buffer update block uses values that are calculated by the frame update.

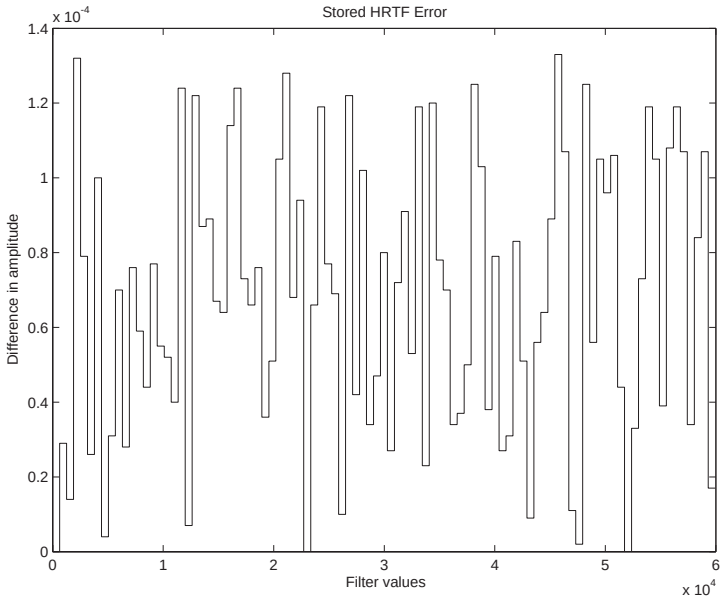

Figure 4.1: HRTF error in database. This figure illustrates the difference between integer and floating point values stored in the HRTF databases. The maximum error is less than $1.4 \cdot 10^{-4}$ (0.01%), which is acceptable.

4.1.1 Frame Update Comparison

The frame update block calculates gain, HRTF, LTD and ITD values. These values are used as basis for further calculations in the buffer update block. The values are written to file after every frame execution in both versions of the code. The buffer update block is active in this test, since the values from the frame update block are only accessed and not modified.

4.1.1.1 Findings

As expected, none of the values were identical in the first run. The differences could be different weights, overflow/underflow problems or/and different functionality between the codes. The most noticeable problem was the completely different HRTF-values. No overflow were present and the calculations were identical. The only noticeable difference was the values in the integer and floating point databases. These databases have information that is loaded to calculate HRTF and ITD values. The integer databases are conversions of the floating point databases and stored in adequate Q-formats. A comparison of values in both databases was performed and confirmed that the integer database contained incorrect values. All integer HRTF databases were immediately reconverted using an adequate Q-format. Comparing a selected integer HRTF database with the respective floating point one gave an acceptable error of 0.01%. The difference in accuracy is shown in figure 4.1.

With the correct set of HRTF values in the database, the test was run again and all values were compared again. The gap between the calculated HRTF-

filter values in both codes became smaller but still it was not acceptable (almost 10%). Tracing back calculations, a bug was found when calculating the weights for azimuth and elevation. The angle weights are needed to interpolate HRTF- and ITD values.

```
azim_weight_high = AzimuthAngle - ((azimuth_offset * A3D_HRFILTER_AZIM_INCR)<<6);

azim_weight_low = azim_weight_max - azim_weight_high;

elev_weight_high = ElevationAngle - ((elev_offset * A3D_HRFILTER_ELEV_INCR)<<6);

elev_weight_low = elev_weight_max - elev_weight_high;
```

The azimuth weights are values that vary from 0.0 to 1.0. The bug made the weight values vary from about 0.1 to 0.9. The error found in this calculation was an incorrect value of the constants azim_weight_max and elev_weight_max. The HRTF filters are stored with an interval of 15 degrees i.e. the filter values stored in the database represent angles like 0, 15, 30, etc in both azimuth and elevation. The incorrect value in azim_weight_max and elev_weight_max, assumed that the interval of the HRTF filters was 16 degrees instead of 15. The correct values for both azim_weight_max and elev_weight_max is 15 stored using the Q-6 format i.e. the value 960. Both constants need to be coded in the Q-6 format since the azimuth- and elevation angles are coded in the Q-6 format.

Next step before running the test one more time, was to analyze other calculations that uses the azimuth and/or elevation weights and to compensate for their new values if necessary. Three such calculations were found: ITD, HRTF Left and Right channel calculations.

```
ITD Calculation
temp =
  ( (((((*low_left_p) * azim_weight_low) +  ((*low_right_p)  * azim_weight_high)) >> 6) *
    elev_weight_low ) + (((((*high_left_p) * azim_weight_low) +  ((*high_right_p) *
    azim_weight_high)) >> 6) * elev_weight_high)
  );

ITD = temp >> 16;


HRTF Left Channel Calculation
temp =
  ( (((((*low_left_p) * azim_weight_low) +  ((*low_right_p)  * azim_weight_high)) >> 4) *
    elev_weight_low ) + (((((*high_left_p) * azim_weight_low) +  ((*high_right_p) *
    azim_weight_high)) >> 4) * elev_weight_high)
  );

HRFilter->h_l[k] = temp >> 16;


HRTF Right Channel Calculation
temp  =
  ( (((((*low_left_p) * azim_weight_low) +  ((*low_right_p)  * azim_weight_high)) >> 4) *
    elev_weight_low ) + (((((*high_left_p) * azim_weight_low) +  ((*high_right_p) *
    azim_weight_high)) >> 4) * elev_weight_high)
  );

HRFilter->h_r[k] = temp >> 16;
```

All three calculations are designed to multiply the weights and, by using shift operations, divide by 1024 keeping as much information as possible. To compensate for the modifications done previously to the azimuth and elevation

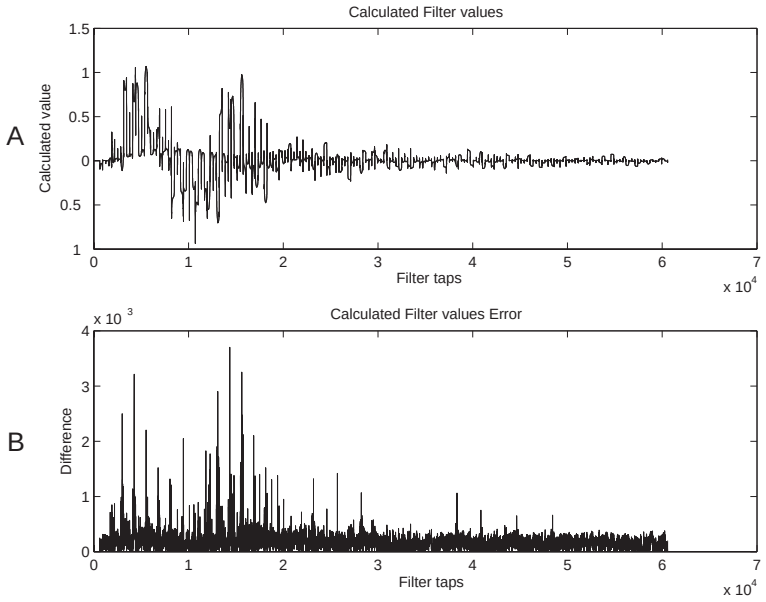

Figure 4.2: HRTF filter error. The top figure shows the first value of every filter block at every frame update in the test. The lower figure is the difference of the integer and floating point values of the calculated HRTF filter block. The maximum error is less than $4 \cdot 10^{-4}$. Such small errors are hardly hear able.

weights, all three calculations must be multiplied by a factor (1024/960) to each weight. Collecting all factors gives a same new value to all three calculations: $(1024/960)^2$. For now, we will only use this value in Q-15 format multiplying it at the last row of all three calculations, since this stage focus on accuracy and precision and not performance. After applying the changes, the test was run again. The calculated HRTF-filter values match with good accuracy and so does the ITD values, see figures 4.2 and 4.3.

The next step, was to analyze the total amount of gain (distance and angular gain). The gain values differed completely in the different codes. A small mistake was found in the script that loaded the distance vector needed in the distance gain. This fault set up two different scene environments making it impossible to match the distance gain values. After correcting the error, both integer and floating point gain values were almost identical. The comparison of the total amount of gain is illustrated in figure 4.4.

Finally, the integer LTD values differed about 25% from the floating point ones. Note that at this point, the older implementation of the LTD calculations was in use. The older Doppler calculations are based on the distance from a source to the listener and the velocities (Doppler distance) of the source and listener. The calculations of the distance between a source and the listener (see section 3.2.4) and the Doppler distance between them (see section A.1), have a small errors. When these large distances are used to calculate the distance change, the error becomes proportionally large. Figure 4.5 shows the difference

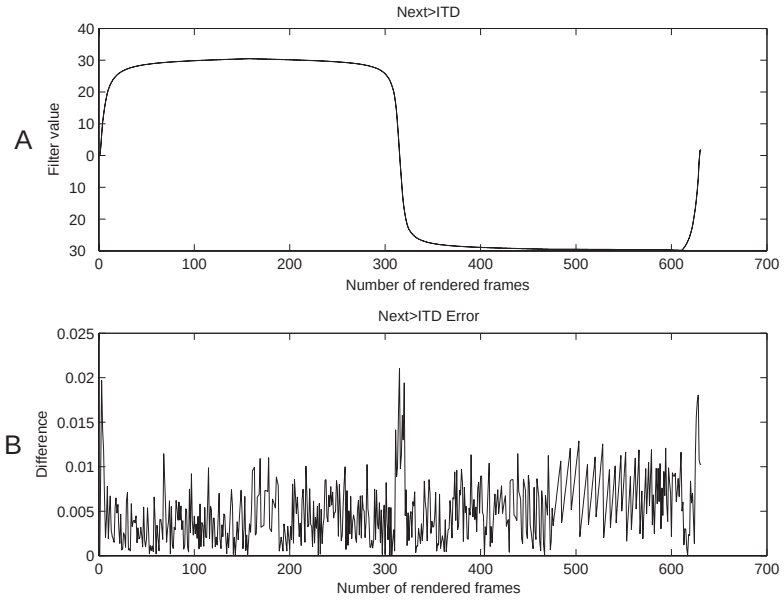

Figure 4.3: ITD error. The top figure illustrates a source rotating around a listener in an oval trajectory and at the same height. The lower figure shows the difference between the calculated integer and floating point values of the ITD. The maximum error is 0.0210.

between the floating point and fixed point accuracy of the distance calculations. The maximum error showed in the figure is about 0.2, 400 meters away i.e. less than 0.1 %. Figure 4.6 shows the difference between the floating point and fixed point accuracy of the Doppler distance calculations. The maximum error for the fixed point Doppler distance calculations is also less than 0.1 %.

Both distance and Doppler distance seem to have sufficient accuracy when compared to the floating point values. However, the next calculation *distance - doppler_distance* is much less accurate, as shown in figure 4.7. The maximum error in this calculation is about 25 %. This error is then projected into the calculated LTD, creating a distortion.

Both distance and Doppler distance calculations involve truncating variables, so that the calculation does not exceed the maximum bit depth of 32-bit. The calculations of distance and Doppler distance are shown in section 3.2.4. The following code shows an example of how values may be truncated in order to preserve a bit depth of 32-bit.

```
TempVar32 =  (Vector16bit[0] * Vector16bit[0]) >> 2;
TempVar32 += (Vector16bit[1] * Vector16bit[1]) >> 2;
TempVar32 += (Vector16bit[2] * Vector16bit[2]) >> 2;
```

By shifting down the product of the vector by 2 bits, the two least significant bits are truncated and a 30-bit value is achieved. This truncation could produce a problem since it is used by both distance and Doppler distance calculations. For instance, if the value 4.9 is truncated to 4 in the distance calculations and the value 5.1 is truncated to 5 in the Doppler distance calculations the result

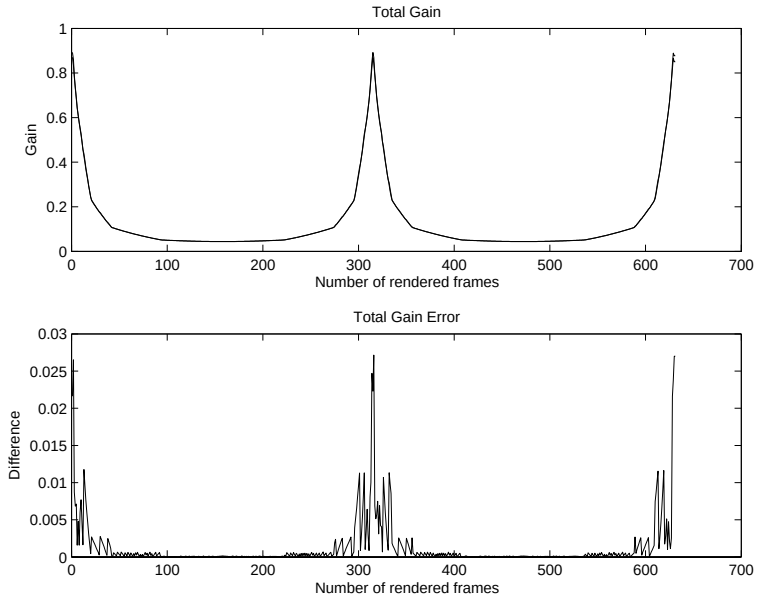

Figure 4.4: Gain error. The total gain mirrors the distance and angular position of a source relative to the listener. In this particular case the source is moving around a listener in an oval trajectory. The maximum difference between the integer and floating point calculated gains is less than 3% (lower figure). This may sound large, but it is acceptable since it means that an object may sound 1.5cm closer/farther away than it is and may sound about 3.8 degrees off its current position.

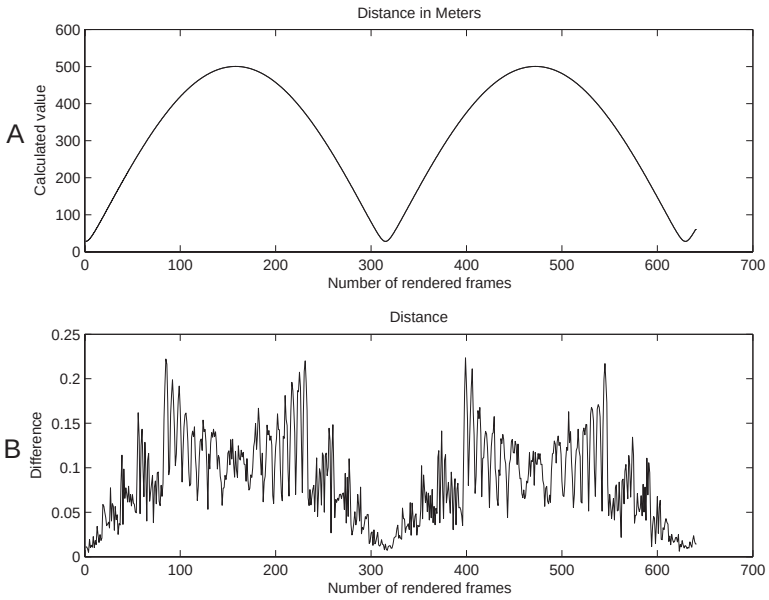

Figure 4.5: Distance error.The figure 4.5A shows a source moving in a oval trajectory 50 meters in height and 500 meters in width. The figure 4.5B illustrates the difference, in meters, between the fixed point and floating point distance positions at a given 3D Audio frame. The maximum error is 0.23 meters at 375 meters, less than 1 %.

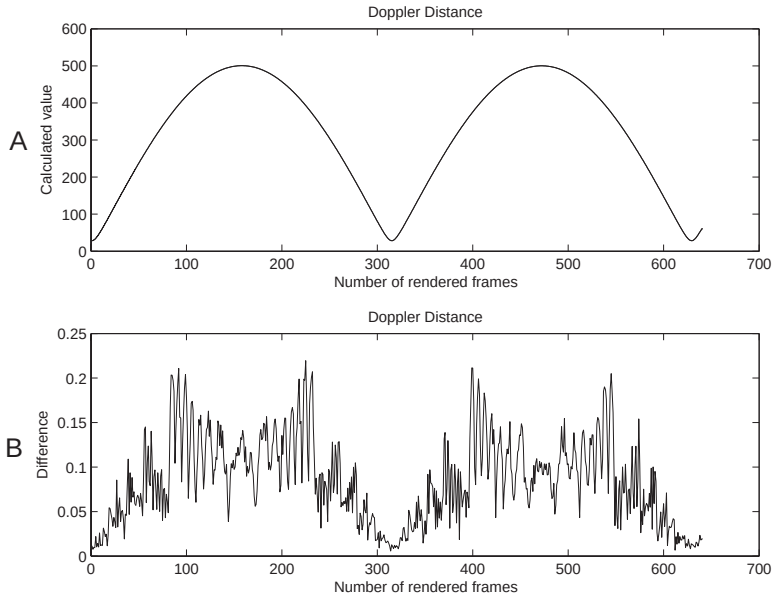

Figure 4.6: Doppler Distance error. The figure 4.6A shows a source moving in a oval trajectory 50 meters in height and 500 meters in width. The figure 4.6B illustrates the difference, in meters, between the fixed point and floating point Doppler distance positions at a given 3D Audio frame. Since the Doppler distance calculations are almost identical to the distance calculations, the error difference between the fixed point and floating point calculations is similar to that of the distance calculations (Figure 4.5). The maximum error in the Doppler distance calculations is 0.21 meters at 400 meters, less than 1 %.

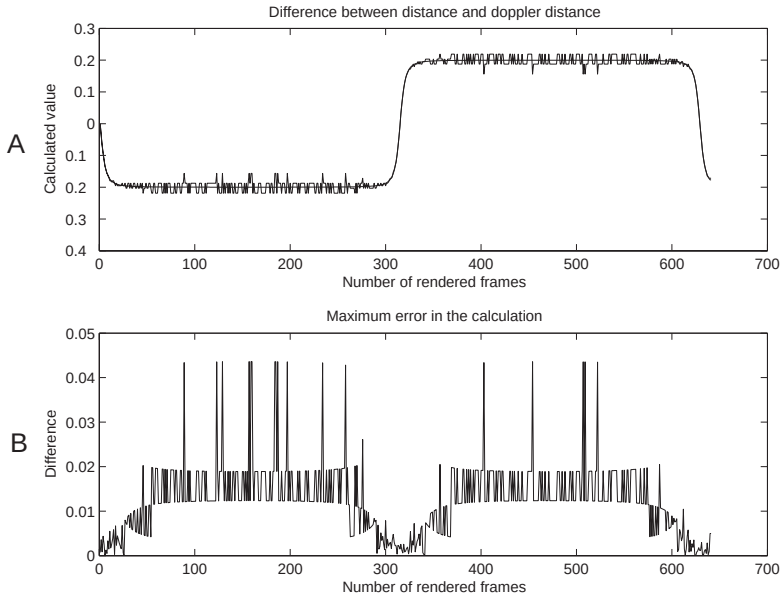

Figure 4.7: Difference between distance and Doppler distance. The difference between distance and Doppler distance is shown in figure 4.7A. The smooth curve, illustrates the floating point calculation and the rippling curve illustrates the fixed point calculations. Figure 4.7B shows the difference between the fixed point and floating point calculations shown in figure 4.7A. The maximum error of $distance - dopplerdistance$ is 0.04 meters at 0.2 meters, about 20 %. To minimize the maximum error in figure 4.7B, the accuracy of distance and Doppler distance must be increased.

would be 1.0 instead of 0.3, an error of 0.7. One approach to solving this problem is to round the values instead of truncating them. Using the same example gives the values 5.0 and 5.0, an error of 0.3. The code below implements rounding without using the conventional round function. By adding half the value before truncation, an efficient fixed point rounding is made.

```
TempVar32 =(
    (ListenerSourceDopplerVector[0] * ListenerSourceDopplerVector[0])+
    (ListenerSourceDopplerVector[0] >> 1)
  ) >> 2;

TempVar32 +=(
    (ListenerSourceDopplerVector[1] * ListenerSourceDopplerVector[1])+
    (ListenerSourceDopplerVector[1] >> 1)
  ) >> 2;

TempVar32 +=(
    (ListenerSourceDopplerVector[2] * ListenerSourceDopplerVector[2])+
    (ListenerSourceDopplerVector[2] >> 1)
  ) >> 2;
```

This new implementation did not reduce the maximum error considerably. An assumption was made that the resolution of the 3D Audio world was not enough. To verify that the precision of a step in the 3D world was enough for the Doppler calculations, the entire Doppler calculation block was rewritten using only floating point calculations. The distance calculations were rewritten as well using floating point calculations, since the Doppler calculations requires the distance between the listener and the source. Since the Doppler/LTD and distance calculations were now identical in both fixed point and floating point codes, the resulting LTD would differ only if the resolution of the fixed point 3D world was not enough. As suspected, the precision of 1/256 meters per step in the fixed point integer world was not enough. The result still differed about 10 %. The precision of the 3D step was altered to 1/32768 which resulted in identical values on the calculated LTD. A 32-bit variable can only hold a maximum size of 131 072 meters, about 130 km, using a resolution of 1/32768 meters per step. A cubic world with a height of 130 km, is a rather small world. The largest 3D world with an acceptable LTD error (less than 1 %) was 4 194 304 meters, about 4 200 km, using a step of 1/1024. While the error was acceptable using the floating point calculations, returning to the fixed point calculations still resulted in an error of about 15%. Now that the 3D Audio step had enough precision, the error must lie within the calculations. There are still two calculations in the Doppler block were precision is lost. The first calculation that loses precision is the ListenerSourceVector calculation. Since the vector has only a 16-bit mantissa, the remaining 16-bit are thrown away. The use of a mantissa and norm is due to a bit depth problem, and is explained in section 3.2.4.

The second calculation that loses precision is the fixed point A3D_sqrt function. The fixed point square root function is described in section 3.2.11. Since A3D_sqrt is a linear interpolation between eight values, a fair amount of precision is lost. However, most of the accuracy is lost in the conversion to a mantissa and norm. To remedy this, a new variable definition must be introduced: the int64. The int64 consists of four 16-bit integers saved as an array. The int64 implementation does not require the conversion into a mantissa plus norm, thus no precision is lost. By converting the int64 into a floating point values and

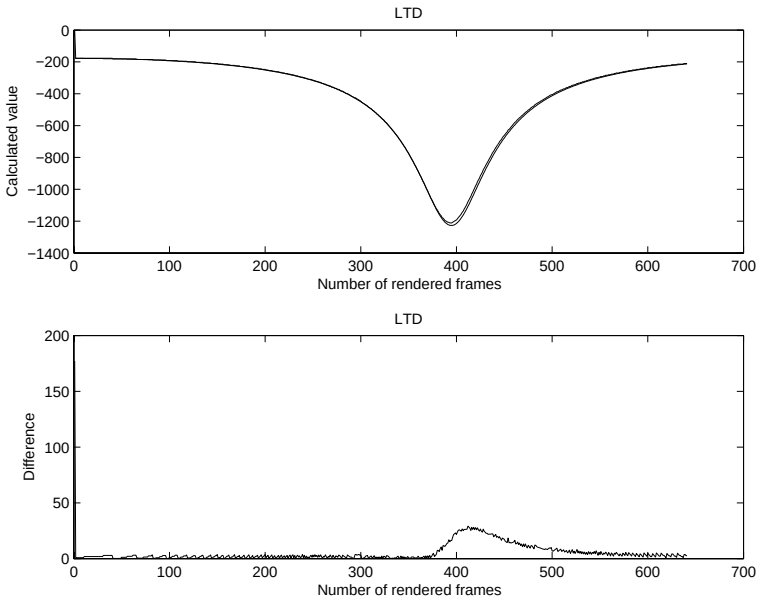

Figure 4.8: LTD values. This figure compares the floating point and fixed point LTD values. The maximum difference is 29 subsamples, about 0.45 samples, when the LTD value is -1230.

using the floating point sqrt function, the LTD error became less than 1 %. An arithmetic 32-bit precision sqrt function was planned to be created as well to eliminate any loss of precision done by a linear approximation. The 64 bit sqrt became too computational complex. Alternative ways of implementing the LTD calculations were explored. A new implementation was created and tested. The new implementation did not have the same accuracy problem as DopplerDistance and Distance, and did not require any 64 bit calculations. The maximum error of the implementation was about 2 %, see figure 4.8. Since the LTD values only affect the pitch of the sound, an error of 2 % is acceptable and does not lower the quality of 3D Audio. The new LTD implementation is described in section 3.2.6.

4.2 Summary

The verification procedure of the 3D Audio engine was described in this chapter. The fixed point 3D Audio engine was compared to a floating point version using the same 3D Audio implementation and configurations. Accuracy errors that were not acceptable were corrected.

Chapter 5

DSP Implementation

This chapter will describe the changes made to the fixed point code developed on a PC, so it could be compiled with code composer and run on a TMS320C55x. This chapter also describes setting up and performing a code profiling, and optimizing the code as a result of the code profiling. Finally a short test is conducted to see that the code works on the DSP-platform. The implementation of 3D Audio on the DSP platform also includes solving the problem on how to create playing sources without reading samples from the PC's hard disk, and how to send the modified 3D Audio samples into a pair of head phones connected to the DSP line out jack.

5.1 Adaptation of the code

After completing the functionality of the 3D Audio engine, excluding reverberation, the code was modified to fit into the C55x.

5.1.1 Code Composer Project

A project in Code Composer Studio is rather complex to create, since it has many parameters to be set. Ericsson provided an older project with default parameter values. Since the old project was also used in with a TMS320C55x, the memory mapping file, a file that configures the memory into blocks, was also copied. The 3D Audio source code files were inserted into the new project. The first build gave some compilation errors and link errors. The link errors, were corrected when the intrinsics library, see section 5.1.6, was included into the source code. The compilation errors were due to the device drive used to play sound in windows, which was now removed using a precompiler directive. Once the errors were corrected, the adaptation of the code started.

5.1.2 Memory Problems

The C55x has a 128 KBytes program/constants memory and a 64 KBytes memory for stack and heap allocations. The heap and stack memory sizes were chosen as follows: System stack: 4.5 KBytes, Stack size: 4.5 KByte and Heap Size: 54 KByte. The enormous stack sizes were chosen to guard against stack overflow problems. The first major problem was to fit the HRTF filter values into the

C55x's memory. The size of the filter values was 46080 bytes. By loading the HRTF values from file and using *malloc* to allocate memory on the heap, gave only 7920 bytes available for the allocation of all other variables in the program. This was not acceptable, since the 3D Audio engine allocates several buffers with a total size of 6720 bytes, a large structure called world using 2656 bytes and 3884 bytes for every source. A new smaller set of HRTF were used. These new HRTF values had only half the amount of values sizing only about 23 KBytes. New ITD values were used as well. The old ITD values, 480 bytes, were replaced by the new ones sizing 240 bytes. Both the new HRTF and ITD values were integrated into the code as constant tables. By doing so, both tables are loaded into the 128 KBytes program/constants memory instead of the 64 KBytes heap memory.

5.1.3 Wrappers

Wrappers were build around commonly used function that differ in the DSP and PC worlds, so that the same code could be compiled in Code Composer and Visual Studio. Wrappers were created around memory allocations/deallocations and around *fread/fwrite*. On the PC-platform *malloc/free* are used and on the C55x the functions are called *DspMem_Alloc/DspMem_Free*. The DspMem_Alloc function allocate the wanted size plus another 32-bits. The extra 32-bits are used to store the amount of memory to allocate. This is used when DspMem_Free is called. DspMem_Free reads how much memory is to be deallocated in the extra 32-bits and deallocates the size plus the extra 32-bits.

```
void * DspMem_Alloc(size_t size)
{
  #ifdef PC_MALLOC
    return malloc(size);
  #else
    unsigned long *iptr;
    iptr = malloc(size + sizeof(unsigned long));
    assert(iptr != NULL);
    *iptr = size;
    return iptr + 1;
  #endif
}

void DspMem_Free(void *ptr)
{
  unsigned long * iptr = ptr;
  #ifndef PC_MALLOC
    if (!ptr) return;
      iptr = iptr - 1;
  #endif
  free(iptr);
}
```

The fread/fwrite are implemented by both platforms, but with different functionalities. The DSP platform can only read/write from file one byte at a time. This differs from the PC, since a PC can read/write large blocks of data from a file. Two wrappers were constructed: *readFromFile* and *writeToFile*. On the PC it only calls fread/fwrite and return/writes the wanted data block. On the DSP

platform it runs a loop reading/writing one byte at a time until the wanted data block is acquired/written from the PC's hard disk. However, one must observe that there is a difference in the way the PC and DSP interpret the byte ordering of variables. The two ways to do this are called 'little endian' representation and 'big endian' representation, where the PC supports the former and the DSP the later. Because of the difference it is necessary to rearrange the ordering of bytes (called byte swapping) when data from an PC platform is to be used on a DSP and vice versa. This is important since the 3D Audio engine reads a wave file from a PC, modifies the samples and then stores them back on the PC. Both readFromFile and writeToFile do byte swapping before passing the data. ReadFromFile is constructed to read blocks larger or equal to 8-bit, to be able to read a wave-file's header. Since the samples are output file is saved in raw format, without a header, writeToFile needs to write blocks of 16-bit or larger and does not support 8-bit writing.

```
int writeToFile(void * ptr, int size, int length, FILE *File)
{
  int16 writing=0;

  #ifdef DSP_C5XX
  int i;
  int byte1, byte2;
  int16 * buffer_p = ptr;

  for (i = 0; i < length; i++)
  {
    byte1 = (buffer_p[i] >> 8) & 0xFF;
    byte2 = buffer_p[i] & 0xFF;

    writing+= fwrite(&byte2, 1, 1, File);
    writing+= fwrite(&byte1, 1, 1, File);
  }
  #else
  writing= fwrite(ptr,size,length,File);
  #endif

  return writing;
}

int readFromFile(void *ptr, int size ,int Length, FILE *File)
{
  int16 reading=0;

  #ifdef DSP_C5XX
    int16 * buffer_p = (int16 *) ptr;
    int16 total = Length * size;

    for (reading = 0; reading < (total >> (size - 1)); reading++)
    {
      buffer_p[reading] =  getc(File);
      if(size > 1)
        buffer_p[reading] |= getc(File)<<8;
    }
    #else
    reading =fread(ptr, size, Length, File);
   #endif

  return reading;
}
```

5.1.4 I/O Routines

When running the 3D Audio engine on the Pre-OMAP board, samples must be read from and to the PC's hard drive. This communication is very slow(one byte at a time) and the DSP is forced to wait until the data transfer is completed. When reading/writing from the PC's hard disk, the DSP could not render more than one 3D Audio frame(10 ms) per second. To be able to optimize and debug the 3D Audio implementation new routines were developed. The new routines, simply defined as FakeIO, did not actually read/write data from/to file. Instead the input routine used wrote zeroes to the requested input buffers to simulate input values and the output routine did not do anything. After these routines were implemented, the DSP could render 3D Audio frames in real-time. The Fake I/O routines were used when profiling or when searching for memory leaks.

5.1.5 Clock Speed

Since the Pre-OMAP development board by default runs with a clock speed of 20MHz instead of its maximum clock speed of 160 MHz, there was a need to set the clock speed. The information in technical specifications and manuals made it possible to set the clock speed with help from an existing function written by Ericsson.

5.1.6 Libraries

The C55x comes with a library called intrinsics developed by Texas Instruments. This library contains a set of functions such as add, sub, mult, negate, etc, that are optimized for the C55x. All functions in intrindefs can be simulated on a PC. The simulation of these functions on a PC may slow down the entire application, on the other hand using these functions on the C55x and setting a high optimizing level in the compiler, results in a performance boost.

5.1.7 Memory Leaks

There were some minor leaks of memory in the 3D Audio engine. They were so small they were not noticeable on the PC, but caused the program to crash on the C55x after about 1 200 frames. The leak was 12 320 bytes after 1 200 frames. To facilitate the search for memory leaks, we assign a variable "totalSize" to keep track of the total allocate memory. Every time DspMem_Alloc was called totalSize was increased by size + sizeof(unsigned long) and when DspMem_Free was called totalSize was decreased by the value stored in the extra 32 bits (stored by DspMem_Alloc). By using printf every time totalSize was altered, it was easy to locate the memory leaks. The entire 3D Audio engine was disabled, enabling one pair of allocation/deallocation procedure at a time. For example, All memory needed by structs was allocated in the allocation function. The deallocation procedure was run directly after the allocation function. The totalSize variable increased to a number of bytes and then decreased to 2894 bytes. This meant that the deallocation of all structures did not deallocate all memory and caused a leak of 2894 bytes. The other memory leaks found were

36 bytes in the initialization/deinitialization process and 8 bytes every 3D Audio frame. The leak in every 3D Audio frame meant that after 1200 frames, 12 seconds of sound, the leak would be 9600 bytes. To be certain that all memory leaks were gone, the 3D Audio engine was set to render 50 000 frames, about 8 minutes and 30 seconds of sound. When the 3D Audio engine had rendered all frames the totalSize variable was 0, i.e. no leaks.

5.1.8 Other Differences

Visual C++ 6.0 was used to develop the 3D Audio engine on the PC-Platform. On the DSP-platform, code composer 2.20 was used. Since Ericsson wished the code to be compatible with Visual C++ 6.0 and code composer 2.20, some issues were discovered. For instance, *sizeof()* returns bytes in the PC-platform, but returns 16-bit words on the DSP. Another important difference is the arithmetic syntax of both compilers. In visual studio multiplying two 16-bit values, adding them to a 32 value and storing the result as a 32-bit variable may look like this:

```
int16 var1 = 256;
int16 var2 = 256;
int32 var32 = 100;
int32 result;

result = var32 + (var1 * var2);
```

Result will be the value 65636. The same code run on the DSP would give the value 100. If two 16-bit value are multiplied together and the product is greater than 32767 or lesser than -32768, the product will be 0. The compiler in code composer multiplies the two 16-bit variables and tries to store the product as a 16-bit value. If the product is bigger than a 16-bit value, the value 0 is written into memory. The correct syntax on the DSP may look like this:

```
int16 var1 = 256;
int16 var2 = 256;
int32 var32 = 100;
int32 result;

result = var32 + ((int32)(int16)var1 * (int32)(int16)var2);
```

By adding the type cast (int32)(int16) before var1 and var2, the compiler translates the type casting as a multiplication of two 16-bit variables, but to store the product in memory as a 32-bit value. This is useful since the compiler does one 16-bit multiplication and a 32-bit addition, instead of a 32-bit multiplication and a 32-bit addition. On the PC platform, Visual C++ will convert var1 and var2 into 32-bits variables, multiplying them together and then adding the product with var32. Since the PC-version does not need to be optimized, but only functional, it is acceptable.

Yet another important issue is the printf arguments. In the PC-platform it is common to use the "%i" flag to print an integer, regardless of bit-depth, to screen. The same flag (or the flag "%d") is used on the C55x to print 16-bit integers to screen. Trying to print a 32-bit integer to screen with this flag will result in a 0. The following code illustrates this problem.

```
int32 i;

for(i = 0; i < 5; i++)
  printf("%i, %ld\n", i, i);
```

The resulting output compiling this code using code composer is as follows:

```
0, 1
0, 2
0, 3
0, 4
0, 5
```

The same code using Visual C++ will print correct values on both columns. To print 32-bit integers on the C55x, the flag "%ld" mus be used. However, if the flag "%ld" is used trying to print a 16-bit variable, the results will be incorrect as well. This is illustrated below.

```
int16 i;

for(i = 0; i < 5; i++)
  printf("%i, %ld\n", i, i);
```

The resulting output compiling this code using code composer is as follows:

```
0, 19770
1, 85306
2, 150842
3, 216378
4, 281914
```

Again, the same code using Visual C++ will print correct values on both columns. This may not seem like a serious problem. However, since the watch window in code composer did not work 100% correctly in the simulator, printf was often used for debugging. When 16-bit variables are printed to screen with values outside the range [-32768,32767], or when all 32-bit variables are printed as zero, and you get completely different values when they are written to file, suspect the printf flags.

5.2 Code Profiling

Profiling is used to study how different parts of the program consumes Central Processing Unit (CPU) time. Profiling is defined as; a technique used to determine how many cycles a processor spends in each section of a program. When a high cycle consuming section is found, it is possible to study what part of that section that is computational complex. Once the part of a section that consumes much CPU is found, optimizations to that part can be made and profiling may be executed again. In this way it is possible to systematic investigate an entire program, different sections or different parts of a program, to find which parts that are consuming most cycles.

5.2.1 Profiling Program

The profiling program used is a part of Code Composer Studio. The most essential information the profiling program gives is; Code size, Execution count, total-, minimum-, maximum- and average cycle consumption including sub functions. Execution count is the number of times a certain section has been called, Total is the summarized cycles consumed of a section, Minimum is the least amount of cycles consumed by a section, Maximum is the largest amount of

cycles consumed by a section, Average inclusion is the average number of cycles consumed by a section. The profiling was run on the DSP with a minimal amount of debugging and using the highest level of optimization, see section 5.3 for details on the optimization settings.

5.2.2 Profiling Results

Code profiling was performed after test and verification of the 3D Audio engine, see section 4. At this stage there existed a fully working 3D Audio engine, without reverberation. The reverberation block can be though as a standalone block. If inserted, it only requires limited modifications to the existing blocks. It was decided that the existing 3D Audio engine was to be profiled and optimized, without the reverberation block.

5.2.2.1 Profiling without Reverberation

Only a handful of the profiled functions will be shown in this section. These functions are: A3D_Execute(the entire engine), A3D_Hrtf_Transform(applies HRFilter), A3D_ResampleSource(resampling of a source), A3D_FrameUpdate(updates positions, velocities, projections, etc), A3D_CalculateCurrentParams (Doppler and ITD buffer length calculations), A3D_GetBuffer (requests buffers), DSPMem_Alloc (allocates memory) and DSPMem_Dealloc(deallocates memory). Throughout all code profiles, the frameUpdate function runs once every 20ms. All functions with the exception of the memory routines and the FrameUpdate are run once every frame (2ms). The functionality of the 3D Audio engine is set to process HRTF, ITD, Doppler and gain. The sample frequency used while profiling was 48 kHz.

Note that the profiling illustrations in this section show the total number of cycles consumed by a function and its subroutines. Since all other functions are subroutines to A3D_Execute, the cycles used by A3D_Execute are the total amount of cycles consumed by the entire 3D Audio engine. To be able to compare optimizing results, profiling was made to the 3D Audio engine without any Code Composer optimization settings. The profiling without optimizations is shown in figure 5.1.

Without any optimizations, the entire 3D Audio engine consumes 433 310 cycles in one frame (2 ms), an estimate of about 217 Mcps. The two most computational complex subroutines in the 3D Audio engine are A3D_Hrtf_Transform, using 298 551 cycles, and A3D_ResampleSource, using 72 944 cycles. Together these two subroutines stand for 86% of the cycles used by the unoptimized 3D Audio engine. Before the next profiling run, the code was optimized as described in section 5.3.1. Figure 5.2 shows the cycle consumption of the optimized code.

The optimized 3D Audio Engine with full code composer optimizations consumes 75 620 cycles, about 38 Mcps. The optimization process has reduced the complexity of the 3D Audio engine by five times. The new consumption of 38 Mcps is below the DSP maximum capability of 160 Mcps. One of the requirements of this thesis is being able to run two playing sources on the DSP. The next profile is run with two playing sources, shown in figure 5.3.

With two playing sources, the consumption is almost doubled from 38 Mcps to about 74 Mcps. This was expected since the A3D_Hrtf_Transform and A3D_ResampleSource are executed for every source. The cycle consumption

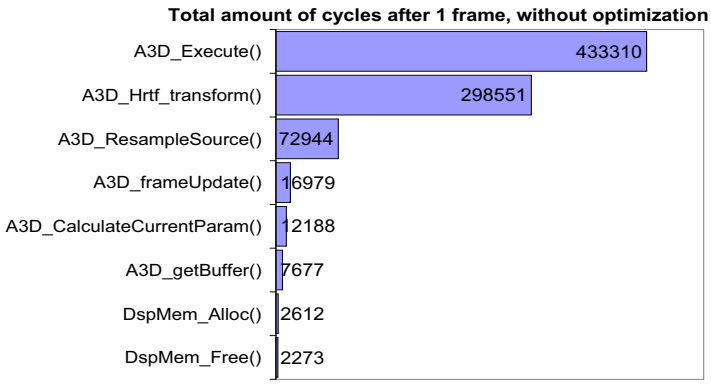

Figure 5.1: The amount of cycles consumed by the 3D Audio engine after one frame with one playing source and no optimizations.

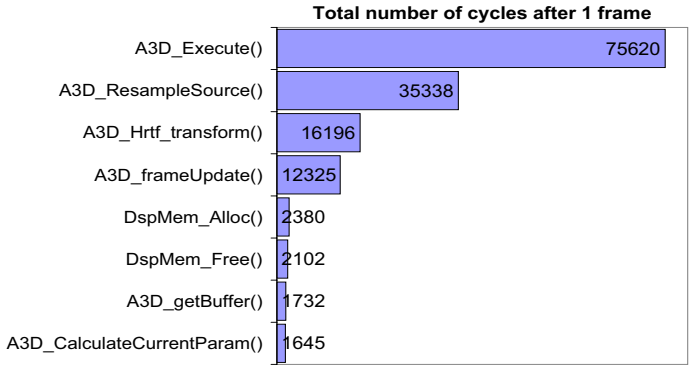

Figure 5.2: The amount of cycles consumed by the optimized 3D Audio engine after one frame with one playing source.

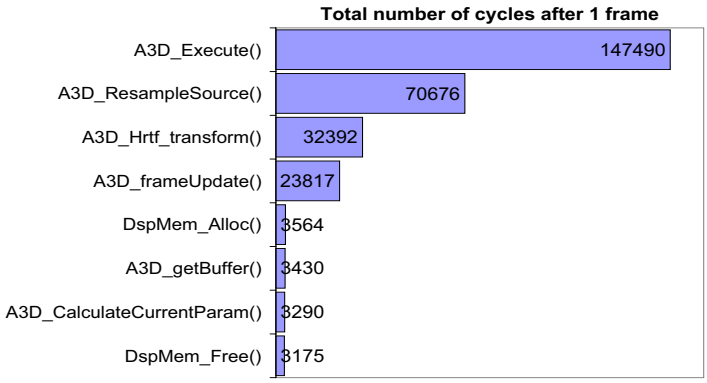

Figure 5.3: The amount of cycles consumed by the optimized 3D Audio engine after one frame with two playing sources.

of the 3D Audio engine with two playing sources, is still below the maximum number of cycles that can be executed on the DSP. To get a more average result using two sources, 25 frames(50 ms) were executed while profiling, see figure 5.4.

After 25 frames, the 3D Audio engine has consumed 3 182 821 cycles, about 64 Mcps. The amount of cycles used by the entire 3D Audio engine dropped from 74 Mcps to 64 Mcps due to the execution frequency of the A3D_FrameUpdate. The A3D_FrameUpdate executes every 20 ms and is only executed twice during the 25 frames, thus reducing the average complexity. Without reverberation, the 3D Audio engine consumes about 40% of the maximum capability of the DSP.

5.2.2.2 FrameUpdate Frequency Comparison

To illustrate the computational influence of the Frame Update a custom profiling was done. Instead of running the Frame Update every 20 ms the Frame update will run as fast as the OutBuffer routine, i.e. every 2 ms. Figure 5.5 shows that the optimized 3D Audio engine, with two playing sources and the Frame Update frequency of 2 ms, consumes about 79 Mcps. Compare this value to 64 Mcps, when the frame update run at a more realistic update frequency (20 ms) see figure 5.4.

5.2.2.3 Profiling with Reverberation

The reverberation implemented depends on the buffers L_o and L_f delivery frequency and when the reverbApply is called.

This is due to that if L_o is smaller than L_f, the reverberation will not be completed by each buffer update call. The Mcps consumption will also for the N_d (Number of reverberation delay nodes) first calls of buffer update be less than

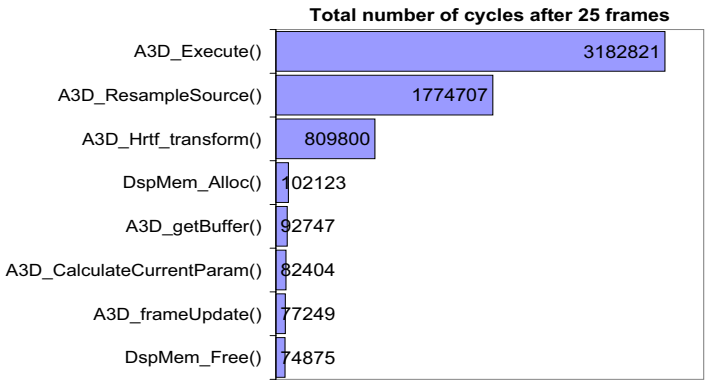

Figure 5.4: The amount of cycles consumed by the optimized 3D Audio engine after 25 frames with two playing sources.

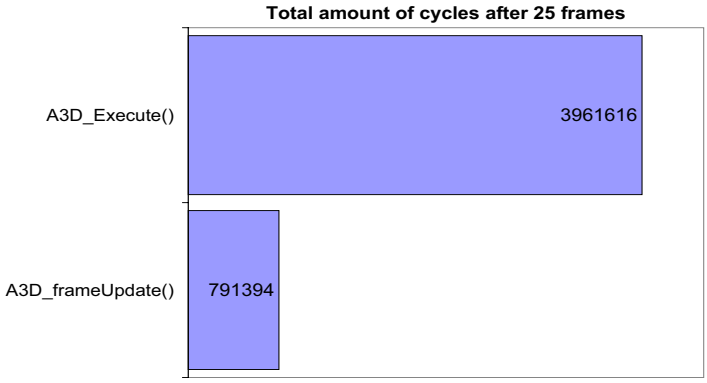

Figure 5.5: The amount of cycles consumed by the optimized 3D Audio engine after 25 frames and running the Frame Update routine every 2 ms.

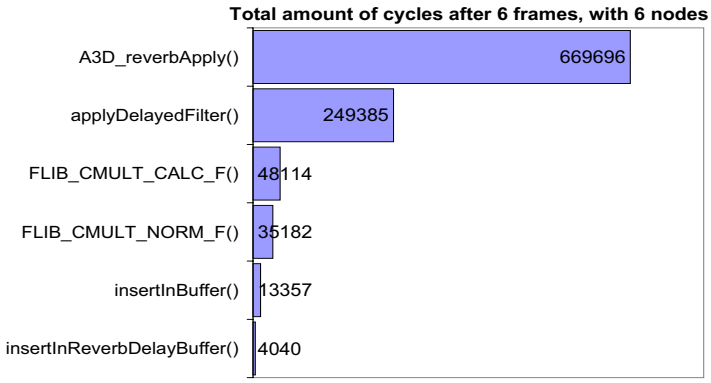

Figure 5.6: The amount of cycles consumed by the optimized reverberation run for 6 frames and with $N_d = 6$.

for the later calls. As shown in figure 5.6 the complexity is 55.8 Mcps contra in figure 5.7 where it is 76.3 Mcps. It is possible to reduce some of the computations by reducing N_d. This is shown in figure 5.8 where the complexity is approximatly 63.9 Mcps. Mainly the reduction comes from the applyDelayedFilter function, where the amount of iterations will be fewer. Therefore the number of calls to the CMult function are reduced. Also, the number of normal compensations are reduced.

5.3 DSP Optimizations

Code composer Studio C/C++ compiler is able to perform various optimizations at different levels: file, program and processor [13]. High-level optimizations are performed in the optimizer, which must be used to achieve optimal code. The optimizer offers different levels of optimization, where level 0 performs simple optimizations like allocating variables to registers, elimination of unused code, etc. Level 3, the highest level of optimization, offers features like loop optimizations, exclusion of all functions that are never called, elimination of global common subexpressions, etc. Note that the highest level of optimization, may remove functions calls or interrupt requests. The code generator performs several additional optimizations, particularly processor-specific optimizations. These optimizations are always enabled but are much more effective when the optimizer is used. The optimizer may also use Program-level optimization. Using Program-level compiles all source files into one intermediate file called a module. The module moves to the optimization and code generation passes of the compiler. Because the compiler can see the entire program, it performs several optimizations that are rarely applied during file-level optimization. For further optimizations, pragma directives may be written into the code to tell

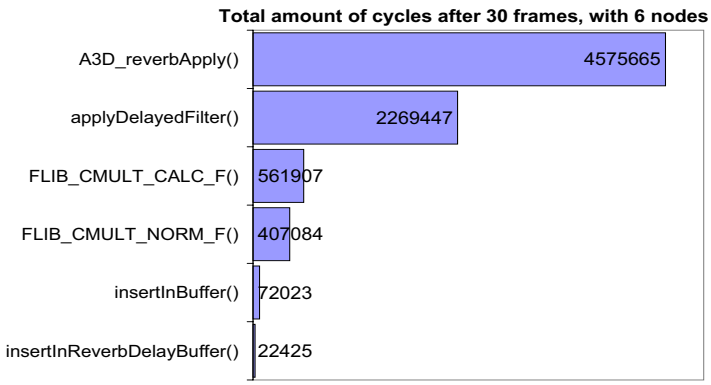

Figure 5.7: The amount of cycles consumed by the optimized reverberation run for 30 frames and with $N_d = 6$.

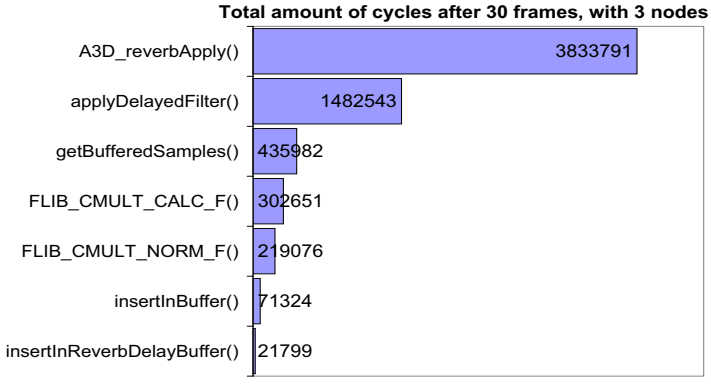

Figure 5.8: The amount of cycles consumed by the optimized reverberation run for 30 frames and with $N_d = 3$.

the compiler's preprocessor how to treat functions. The pragmas used in this thesis were: MUST_ITERATE and UNROLL. The MUST_ITERATE pragma specifies to the compiler certain properties of a loop. Through the use of the MUST_ITERATE pragma, you can guarantee that a loop executes a certain number of times. The information provided by this pragma helps the compiler to determine if it can generate a hardware loop (localrepeat or blockrepeat) for the loop. The pragma can also help the compiler eliminate unnecessary code. The next subsection describes the parts of the 3D Audio engine that were optimized.

5.3.1 Optimizing the Code

After the first profiling run, see section 5.2.2, there was a need to optimize the HRFilter- and Resample functions. The HRFilter block alone, consumed 150 Mcps almost the entire capacity of the DSP. Most of the HRFilter calculations used several operations like additions, multiplications and shifts. In the intrindefs library supplied by TI, there is one operation called L_mac (uses the DSP MAC). The MAC instruction multiplies two variables and shifts the product left by 1. The result is added to a third variable and saturated. The HRFilter calculations were modified to use the MAC instruction. Since the HRFilter loops for every sample in a playing source, the benefit of using the MAC instruction instead of separate operations is considerable. The HRFilter cycle consumption dropped from about 150 Mcps to 53 Mcps, without the help of the Code Composers optimizer. The intrinsics library also defines mapping to simple arithmetic operations. Using these mapped functions instead of C syntax facilitates the optimization procedure of the Code composer optimizer. These function were found to be faster in 32-bit calculations, but the default 16-bit arithmetic operations were faster than the mapped intridefs ones. This was tested by replacing all 16-bit calculations in one specific function (resample function) with intridefs, and profiling one 3D Audio frame. If the profile resulted in lesser cycles, the modification was kept. The same procedure was made with 32-bit calculations. All 32-bit calculations in the 3D Audio engine were modified to use the mapped intridefs operations. Pragma directives were written to all loops to get hardware loops if possible, as described in the previous section. All calculations were also analyzed using only 32-bit operations when needed. Unfortunately, no documented profile was made after modifying the code with intridefs operations, i.e. the code was profiled and it ran faster, but the numbers were not saved.
There was no need for rewriting 3D Audio blocks into assembly, since the optimizations described above were more than enough to meet the requirements. By optimizing the code using Code Composer's optimizer, the amount of cycles consumed by the 3D Audio engine were reduced 5 times, see section 5.2 for details.

5.4 Audio input and output

Using stdIO routines for input and output of samples is slow on the DSP platform, see section 5.1.4. Using these functions to get samples for playing sources would make it impossible for the 3D Audio engine to run in real-time. The so-

lution on how to create playing sources without reading samples from the PC's hard drive and how to send modified 3D Audio samples to a pair of headphones, was the DSP's audio codec. The audio codec, TI TLC320AD77C, is a 24 bit, 96 kHz stereo codec. The audio codec has a stereo line in and a stereo line out. The codec is connected to the DSP via one of the Multichannel Buffered Serial Port (McBSP)s. It is possible to set up the McBSP to return an interrupt when it has retrieved a specific number of samples (1 - 256). The McBSP is set to retrieve 16-bit words, these words will be stored by the McBSP in a FIFO until copied to memory in the interrupt. There is also the possibility to use DMA to transfer data from the McBSP to a specified memory location. In this way it is possible to get an interrupt from the DMA controller when transfer to memory is complete, and thereafter to process the samples located in the memory.

5.4.1 Setting up McBSP

In order to setup the McBSP for continuous reading from the Codec, a number of registers and jumpers most be set. The Code Composer Studio contains a number of examples, the first step was to search theses examples for any useful knowledge. This was found not to give any usefully information, since the examples shipped with the development package, was designed for another Audio Codec. After consulting the supervisor, an example package for the correct board was retrieved from TI. In this package there were one example for the use of the Codec. The example, read one sample from each input channel via interrupt and did output of these samples directly to line out. This example seamed to be exactly the right start, though when creating a new Visual Studio project, building the source and execute it, the example was found to be non functional. Since the code had been written by TI, it was thought of as being at least almost correct. The next step was to analyze why the code did not work. To begin this process first the jumpers on the Pre-OMAP card were checked to have the correct settings. The correct jumper settings was applied. The next step was to analyze the register settings used in the example. The fault was thought on as perhaps being originated from that for example the wrong buffer size was used. Based on this assumption the register settings was controlled, but no errors could be found. The structure of setting up the McBSP was compared with the structure supplied by the TMS320VC5510 Data Manual [9], and found to be correct. Though when the initial register setting, i.e. the disabling of the McBSP, was checked it was found not to set the McBSP to disabled, rather to enable the McBSP and thereby disabling all possibilities to set any of the McBSP registers. Several manuals was used to be able to set Jumpers and Registers; TMS320VC5510 Data Manual [9], TMS320VC55X Peripherals Reference Guide[10], L301/5510 Hardware Specification [11], TLC320AD77C 24-Bit 96 kHz Stereo Audio Codec Data Manual [15] and TMS320VC5501/502/5509/5510 DSP Multichannel Buffered Serial Port (McBSP) Reference Guide [16]. Enabling of the McBSP, required a single bit to be set in a register, to be able to do this without destroying the rest of the register an assembly macro was created. The macro read the register values set one of the bits, and wrote back the new value.

5.4.2 Setting up the DMA Controller

According to the manuals each DMA controller can move date from one memory area to another. This means that in order to uses DMA for the McBSP, there is a need of using DMAs, one per channel. Each DMA will be set to have its source to each of the channel registers at the McBSP. The memory destination for the DMA needs to be aligned in memory. Each DMA will have its own destination area. The Memory addresses set in the DMA registers need to be given in byte addressing, not in words which is the normal mode for the DSP. Also for DMA there existed a couple of examples shipped with the board, though most of these examples used digital loopback mode to simulate transfers. The examples worked as a structural guide on how the DMA should be set up. The manuals used to set the registers are; TMS320VC5510 Data Manual[9] and TMS320VC55X Peripherals Reference Guide[10].

5.4.3 Combining McBSP and DMA

In order to make the DMA controller transfer data from the McBSP to the memory, the interrupt used for the McBSP must be disabled. And the DMA must be configured to listen to the McBSP and be enabled. The DMA must also be configured on how much data it shall transfer when called upon.

5.5 Summary

This chapter described all alterations done to the working 3D Audio engine so that it could compile and run on a DSP. Code profiling was performed on the 3D Audio engine, computational complex blocks were optimized and Code Composer Studio was set to the highest level of optimization. The DSP was also set to sample sound from the line in jack, to modify samples using the 3D Audio algorithm and to send the modified samples to the output jack, where a pair of headphones were connected.

Chapter 6

Test of 3D Audio on DSP

The DSP platform has several advantages over the PC platform. When the 3D Audio engine is optimized (without reverberation), the DSP can render 3D Audio frames three times faster than real-time, considerable faster than both PCs used in this thesis. A major drawback is that if the DSP uses any I/O (read/write) to the PC's hard disk, the performance becomes 100 times slower than real-time. Testing the 3D Audio implementation was done in two tests: Execution and Accuracy verification. The execution verification is done to verify the execution of the 3D Audio engine, i.e. simulating 1 hour of play and checking memory allocations or unexpected behavior like CPU crashes. The accuracy verification uses the I/O functions and renders one minute of sound. The output Wave file is loaded into Matlab and compared with a reference wave file. Note that all DSP optimizations are turned on during this phase and that the DSP audio codec, section 5.4, was still not implemented at this stage.

6.1 Execution verification

The test ran for 20 minutes rendering 1 800 000 frames which corresponds to 1 hour of play. Using only one playing source, with effects like HRTF, ITD, Doppler and gain the 3D Audio engine managed to keep the maximum allocation size below the requirements. The maximum allocation size denotes the maximum number of bytes requested by the 3D Audio engine during its execution. During this test no actual input was used and no output was created. The requested input buffer consisted of a set of zeroes. The input buffer, containing only zeroes, was processed and modified using the 3D Audio engine. The output samples produced by the engine were not sent and saved to the PC's hard drive. Eliminating all I/O routines still allowed the simulation of the execution of 3D Audio, but at a considerable greater speed. One hour of samples was generated without any unexpected memory allocations or CPU crashes.

6.2 Accuracy verification

The accuracy test rendered 30 000 frames, a one minute long Wave file, on the PC's hard disk. A reference wave file was rendered on the PC platform, using the same fixed point 3D Audio engine and configurations, and both output wave

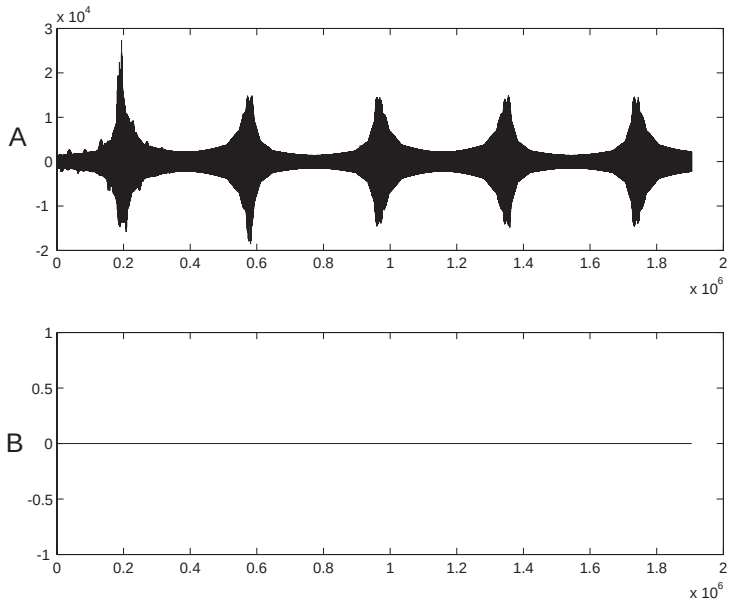

Figure 6.1: The figures show the left channel output signals (6.1a), the first generated on the DSP and the other on the PC platform. The difference between the signals is always zero, shown in figure 6.1b, thus the signals are identical.

files were compared in Matlab. The fixed point 3D Audio engine is compared to itself, since it is already determined that the 3D Audio engine behaves almost identical as the floating point version on the PC platform. Since both the PC and DSP platforms use the same version of the fixed point 3D Audio engine with the same configuration, the output values must be identical. Figures 6.1 and 6.2 illustrate the left and right channel output signals generated on both platforms, and the difference between the signals.

Both output channels on the DSP platform are identical to their respective channel on the PC platform. Since the output difference between the platforms is zero, the accuracy of the DSP platform must be equal to the accuracy of the PC platform when using the same version of the code, even with DSP optimizations turned on.

6.3 Summary

This chapter described the verification procedures done to the 3D Audio engine after all DSP optimizations were completed.

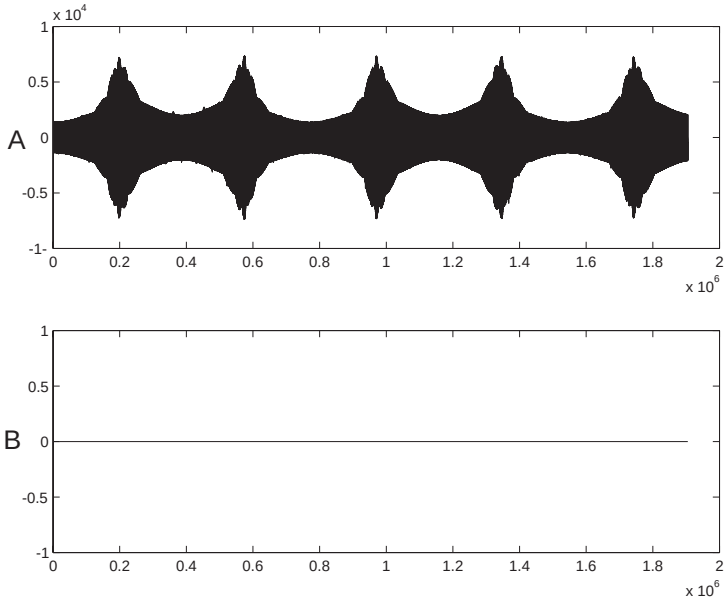

Figure 6.2: The figures show the right channel output signals(6.2a), the first generated on the DSP and the other on the PC platform. The difference between the signals is always zero, shown in figure 6.2b, thus the signals are identical.

Chapter 7

Discussion

The purpose of this chapter is to give the reader further knowledge on different approaches on three chosen problems: development strategy, algorithm implementations and methods of accuracy verification. The challenges within these topics have influenced this thesis the most. The development strategy was chosen at an early stage to start the programing process as quickly as possible (see section 7.1). After examining the existing floating point code, it was decided to use a completely new implementation algorithm (see section 7.2). A floating point code was developed by Ericsson using the fixed point engine as base. The floating point version of the 3D Audio engine was developed to verify/compare the accuracy of the fixed point 3D Audio engine implementation (see section 7.3).

7.1 Development strategies

The development strategy of this thesis was to develop as much as possible on the PC platform, due to its "infinite" resources, and then to modify and optimize the source code to fit the DSP platform. Another strategy could be to develop directly on the DSP platform. Both strategies have their advantages and disadvantages. For instance, developing directly on the DSP platform does not require any additional alterations to run on the DSP. On the other hand, other problems like limited memory and CPU cycles, I/O speeds, limited documentation, etc, become present under the development stage. On the PC platform, a disfunctionality of a program, is likely to be caused by a bug in the program. Developing on a DSP, a disfunctionality may be caused by a source code bug, memory mapping errors, communication problems, hardware problems, etc. However, a programmer experienced in developing directly on DSP platforms should learn to recognize the behavior of a DSP, thus developing much more efficient than a person with no or little experience of DSP platforms. A third strategy could perhaps favor both the experienced and the inexperienced DSP developer. By dividing the program into modules and developing them on the PC platform, removes resource limitations and other DSP related problems. When a module is complete, it is altered to run under a DSP platform. The time needed to modify the program in order to make it run on a DSP platform, is spread out during the development stages. Smaller modules are less complex

than an entire implementation, making the conversion into a DSP platform easier. A drawback of this development strategy is that implementations based on several modules, may be less efficient than an implementation without modules. Though advantages as easy replacement of code sections make module implementations in many cases a good choice.
Another strategy would be to pick a cross platform language like JAVA. JAVA is both object oriented and cross platform. Many mobile devices, cellular telephones, PDA, etc, already support JAVA. JAVA allows the same implementation to be executed on any platform supporting a compatible JAVA version. Unfortunately, the JAVA programming language is not supported in native mode by embedded systems. This means that a JAVA implementation of an algorithm is not as fast as, for instance a C or a Fortran implementation of the same algorithm.

The strategy chosen at the start of a development process was to develop as much as possible on the PC and then to modify the code to run on the DSP. In the final stage we believe that this was the best choice at the start of the project. The actual strategy used is a combination of strategies, the strategy to first develop on a PC platform and then modify the code for the DSP and the strategy to implement the code in modules. That is first to create the flow where modules depends on each other. There after to write the code in such a way that it is possible to both move modules between the PC and the DSP platform as well as remove a module or replace a module with a simplified version.

7.2 Algorithm implementations

Many developers prefer to code their own programs from scratch. If all algorithms are known and descriptions/documentation is available, programmers could develop a more efficient code starting from scratch rather than using a pre existing code. Algorithms may be implemented in different ways, yielding that it is difficult to code a completely new efficient implementation based on a pre-existing code. Some parts of the implementation will most likely be similar to sections of the pre-existing code. When creating a program without any reference code, it can be easier to design the flow and structure of the new implementation. Designing the flow of a program forms the entire implementation and determines the efficiency of the program. To take advantage of an existing code, the structure and/or flow of the new implementation must resemble that of the existing code. The efficiency of the new implementation lies in the developer's ingenuity, when no existing code is available. Using an existing code speeds up the development process, but the new implementation's efficiency is less dependent on the developer. This is perhaps wanted by smaller companies who do not have sufficient resources to research and implement their own programs.

7.3 Accuracy verification

On embedded systems, a fixed point implementation of an algorithm is less computational complex than a floating point implementation of the same algorithm. A common problem developing a fixed point program is to verify the accuracy.

If time and money were no concern, the best way to get around the accuracy verification problem would be to create a floating point implementation based on the fixed point code. In this way developers could easily compare key variables and verify the accuracy and quality of their fixed point implementation. This of course a non-economical approach. A company would not want to develop two implementations of the same program, if there is no profit involved. Considering the economical and workload aspects, the best way a developer could verify the accuracy of a fixed point code is to use existing software. For instance, Matlab can be used to plot key variables and using common sense and knowledge, verify the different parts of the fixed point implementation. The use of a calculator, pencil and paper might in many cases be also enough.

Chapter 8

Conclusion

This thesis describes the implementation, optimization and validation of a fixed point 3D Audio model for a DSP. The work has been divided in to three main sections; implementation of fixed point code on a PC, implementation and adaptation of the fixed point code on a DSP and optimization of the code.
Each section has be done in segments. In the implementation of fixed point code on a PC section, the main segments has been; HRTF, ITD, Resampling, Distance and angular gain, LTD and Reverberation besides from these segments, program flow and buffer handling has been implemented. In the implementation and adaptation of the fixed point code on a DSP, the main segments has beside from adaptation of the segments from the first section been to verify the same results exists on the DSP as on the PC implementation. In the optimization of the code, the code from the second section has been optimized mainly for CPU usage but also in some level for memory usage.

Throughout all segments of each section, the code implemented has been verified and when possible compared to a floating point version, created in parallel by Ericsson. The verification has both been done for each section, for each segment and for all segments working together.

During the project, many problems has been found, identified and dealt with. The work has been completed with great satisfaction and the progress has been almost according to schedule decided at the start of the project.
The result, a according to specifications fully functional 3D Audio demonstrator.
Due to time demanding development stages in the latter sections of the work, such as being able to feed the DSP with samples in realtime, the demonstrator is not fully optimized. In section 8.1 further possible optimizations and developments are pointed out.

8.1 Further development

Further development of this project mainly consist of possible optimizations both on memory and on CPU usage. The most cycle consuming part is the reverberation, which now is implemented in frequency domain, giving approx-

imatly the same memory and the same Mcps cost for any amount of sources. This implementation gives a good reverberation effect but the cost is rather high. Another more simple implementation, in time domain, would reduce the quality slightly but would also cost less amount of cycles but with a slightly higher memory consumption. Next to the Reverberation, also consuming much resources, is the resample function. Optimization for the resampling function can be done in two ways. Translating and optimizing the resample function into assembly code, enabling usage of parallelism, or to implement the resampling function into a hardware solution. A hardware supported resampling function would make it possible to both perform resampling without putting load on the DSP. The HRTF transform, consumes half the amount of resources as the resample function. The cost of the HRTF could be lower, with an assembly implementation. Where it would be possible to control usage of parallelism and usage of MACs.

Some general memory optimizations could be done, by further usage of in-place buffer and array operations. To reduce the cost of both the actual sampling interrupt and the unnecessary amount of interrupts created. The transfer from McBSP to memory could be handled by the DMA controller. And the input and output samples should be grouped in some way, in order not to get a interrupt per sample.

Bibliography

[1] " http://www.dolby.com/dolbyheadphone/". *Dolby Headphone Technology.* August 2003.

[2] American National Standard for Information Systems. *Programming Language C.* American National Standards Institute (ANSI X3.159-1989).

[3] Jens Blauert. *Spatial Hearing, The Psychophysics of Human Sound Localization. Revised Edition.* The MIT Press, 1997.

[4] Durand R. Begault. *3-D Sound for Virtual Reality and Multimedia.* Ames Research center, 2000.

[5] " http://focus.ti.com/docs/prod/productfolder.jhtml?genericPartNumber= TMS320VC5510". *TMS320VC5510* August 2003.

[6] Texas Instruments. *310/5510 Pre-OMAP Hardware Development board, Revision 1.* Texas Instruments, 2000.

[7] " http://www.portaudio.com/". *PortAudio - portable cross-platform Audio API.* August 2003.

[8] " http://java.sun.com/". *Java 3D API.* Summer 2003.

[9] Texas Instruments. *TMS320VC5510 Fixed-Point Digital Signal Processor, Data Manual.* Lit. number SPRS076F, Texas Instruments, 2000.

[10] Texas Instruments. *TMS320VC55x DSP, Peripherals Reference Guide, Preliminary Draft.* Lit. number SPRU317B, Texas Instruments, May 2001.

[11] Texas Instruments. *L301/5510 Pre-OMAP Board, Hardware Specification, Revision 1.1.*, Texas Instruments, Sep 2000.

[12] " http://www.hal-pc.org/ clyndes/computer-arithmetic/floats.html". *standard IEEE 754* September 2003.

[13] Texas Instruments. *TMS320C55x Optimizing C/C++ Compiler UserŠs Guide* Lit. number SPRU281D, Texas Instruments, July 2002.

[14] Patrik Sandgren. *Implementation of a development and testing environment for rendering 3D audio scenes* Royal Institute of Technology, August 2000.

[15] Texas Instruments. *TLC320AD77C 24-Bit 96 kHz Stereo Audio Codec. Data Manual* Texas Instruments, 1999.

[16] Texas Instruments. *TMS320VC5501/502/5509/5510 DSP Multichannel Buffered Serial Port (McBSP) Reference Guide* Lit. number SPRU592A, Texas Instruments, December 2002.

Appendix A

Doppler Implementation Problems

During the thesis, accuracy and complexity problem were encounter in the Doppler calculation block, see section 4.1.1 for more details about the problems and solutions. The problems were solved by replacing the Doppler block with a new implementation. This new approach was equally efficient in terms of computational complexity and is described in section 3.2.6. The first implementation is described in the next section.

A.1 The First Doppler Implementation

The LTD is calculated in every frame, the value depends on the distance to the source and the speed of sound in the medium that the source and listener are located in. For each frame a new value is calculated and the older value becomes the previous value, giving next- and previous LTD values. The LTD calculations consists of calculating a distance called *DopplerDistance* and is based on the current positions and velocities of the listener and a source. The current distance between the listener and a source is also calculated Subtracting DopplerDistance to "DistanceToListener", which is based only on positions, gives the distance changed by the velocities of the listener and the source. The LTD is calculated by multiplying the difference of DistanceToListener and DopplerDistance with the inverse speed of sound, this gives the time it will take the sound to travel this distance. This would correspond to the duration of the sound that is on its way in the air.

The duration is then multiplied with the sampling rate of the source to convert the amount of time into samples. Finally, the LTD is obtained by dividing the number of samples with the number of buffers between frames.

$$\begin{aligned} t_f &= \frac{N_b \cdot L_o}{S_r} \\ LSDV &= P_s + (V_s \cdot t_f) - P_l - (V_l \cdot t_f) \\ DopplerDistance &= ||LSDV|| \end{aligned}$$

Table A.1:

Symbol	*Description*
N_b	denotes the number of buffers per frame.
P_s	denotes $Source \rightarrow Next \rightarrow Position$
P_l	denotes $Listener \rightarrow Next \rightarrow Position$
V_s	denotes $Source \rightarrow Next \rightarrow Velocity$
V_l	denotes $Listener \rightarrow Next \rightarrow Velocity$
V_{so}	denotes SpeedOfSound
t_f	denotes Time between frames
$LSDV$	denotes ListenerSourceDopplerVector

$$LTD = \frac{(DistanceToListener - DopplerDistance) \cdot V_{so}^{-1} \cdot S_r}{N_b}$$

Notation accordingly to table A.1. TimeBetweenFrames, denotes the time between every frame update in seconds. ListenerSourceDopplerVector is the distance of every axis from the listener to the source, taking in consideration the velocity of listener and source. The calculation of the ListenerSourceDopplerVector is similar to the ListenerSourceVector in the distance calculations (section 3.2.4).

As described in section 4.1.1, the difference between DistanceToListener and DopplerDistance did not have enough accuracy. The maximum error was about 25 %!.

Appendix B

Matlab Scripts

The Matlab scripts used throughout this thesis were developed to compare the accuracy between the floating- and fixed point versions of the 3D Audio engine, setting up parameters in the 3D engine and for debugging reasons. By defining a flag in the fixed point or floating point version of the 3D Audio engine, forces it to load a file from the hard disk. This file is generated by a Matlab script. Inside Matlab several key parameters like source movement, scene characteristics, etc, are set manually. When the file is generated, both versions of the code loads the file and are configured identically. This is done to create exactly the same characteristics when comparing both versions of the 3D Audio engine.
When both codes run with the *Matlab flag* set, they start writing parameters to file. The files contain information about key variables inside the engine, output signal, input signal, etc, for every frame. The comparison script loads up the all data from both the floating- and fixed point files and plots 2 subplots for every key variable. The first figure contains a plot of both the fixed point and floating point values. The next subplot plots the difference between the fixed and the floating point values. A maximum of 20 figures(depending of which values are activated), with 2-3 subplots, are plotted. There are other smaller scripts used for debugging, they plot specific variables. The smaller scripts are useful since they use considerable less amount of time and memory than the big comparison script(the smaller scripts do not plot as many variables as the comparison script).
These scripts made it easier to compare or debug the 3D Audio engine. Since the code could easily be moved between the DSP and the PC platforms, any optimization made in the DSP platform could be verified in Matlab. In conjunction with the supervisor of this thesis it was decided that the source code of the Matlab scripts will not be published.