CARNEGIE MELLON UNIVERSITY

FINAL PROJECT REPORT 18-551, FALL 2007

Lose Yourself

A Virtual Reality Audio Immersion

Group 10

Crystal Lee (crystall@andrew.cmu.edu)

Laura Pritchard (lpritcha@andrew.cmu.edu)

Liam Bucci (lbucci@andrew.cmu.edu)

12/10/2007

Table of Contents

Introduction

The Problem

Virtual Reality (VR) systems, today, have become much more common sought after and researched technologies that are rapidly expanding in complexity and breadth of use. They have applications in entertainment, education, and medical instruction, to name a few. In addition to virtual reality video games, these technologies are utilized for instructive purposes in training doctors, pilots, and other professionals whose preparation normally requires expensive simulation. All of these applications demand an even greater level of user interaction. Audio cues in these environments are crucial features to making artificial environments true to form. When a person turns their head, they expect both their visual field to change as well as the relative direction of incoming sounds. It can be very disorienting for a user to have mismatching visual and auditory information. This expanding market for immersive environments is the motivation behind developing real-time systems capable of dynamically reacting to a user's position relative to his environment and to engage more of his senses than just sight.

There are several key indicators utilized by acoustic rendering systems that enable a user to track sound in a spatial environment. First, the location of sound sources relative to the receiver creates differences in the way sound propagates to the right and left ears. This effect can be modeled by head related transfer functions, the recorded frequency response of sound reaching an individual's ears. In room environments, secondary considerations include the early reflections and late reverberation; that is the higher order reflections in a room. Each individual

room geometry and its wall properties will uniquely determine the acoustics of an enclosed space. The absorption coefficients of each surface and air allow us to derive the amount of reverberation and dissipation of sound in the room. There exist several models that can simulate these effects fairly accurately.

Prior Work

There has been a lot of research done on this particular problem and many of the implementations work very well at mimicking actual auditory environments. At the same time, many of them require a larger amount of computing power than we have available for this project.

With regards to previous 18-551 work, Group 2 of Spring 2001 also did audio processing using head related transfer functions in their project, "Music to the Ears: Creating Multi-Dimensional Sound for a Virtual Environment." The entirety of their project consisted in implementing HRTFs to render a virtual audio environment while ignoring the discrepancies that would arise from different environments and not just in the relative positions of the listener and the sound source. Upon its conception, our project had the novelty of accounting for different possible environments by doing room modeling and accounting for reverberation. This would have made it much more computationally expensive because it would have involved applying HRTF filtering for each reflection as well as doing the Schroeder Reverberation calculations and integrating them into the final signal. This would also have meant that our code would have had to be more efficient and optimized than that of the previous group if we wanted to implement it in real time. Our implementation would have more accurately represented the real environment.

However, due to certain unavoidable circumstances pertaining to our original group's dynamic we were unable to implement the room modeling and late reverberation models. Given that the core of the project consisted of implementing the HRTF's and that we encountered some problems in implementing the FFT and IFFT (which will be discussed later), we were unable to pick up the slack with regards to the room modeling and Schroeder reverberation calculations when it became clear that they would not be done otherwise. The end result is that our implementation achieved nearly the same purpose as Group 2's project in 2001. One notable difference is that Group 2 made use of the codec while we output our signal through the PC sound card. It should also be noted that they utilized HRIR's from the MIT Media Lab's KEMAR Dummy-Head Database, whereas we used a portion of the UC Davis CIPIC HRTF Database- a database whose measurements were taken from real subjects.

Our Solution

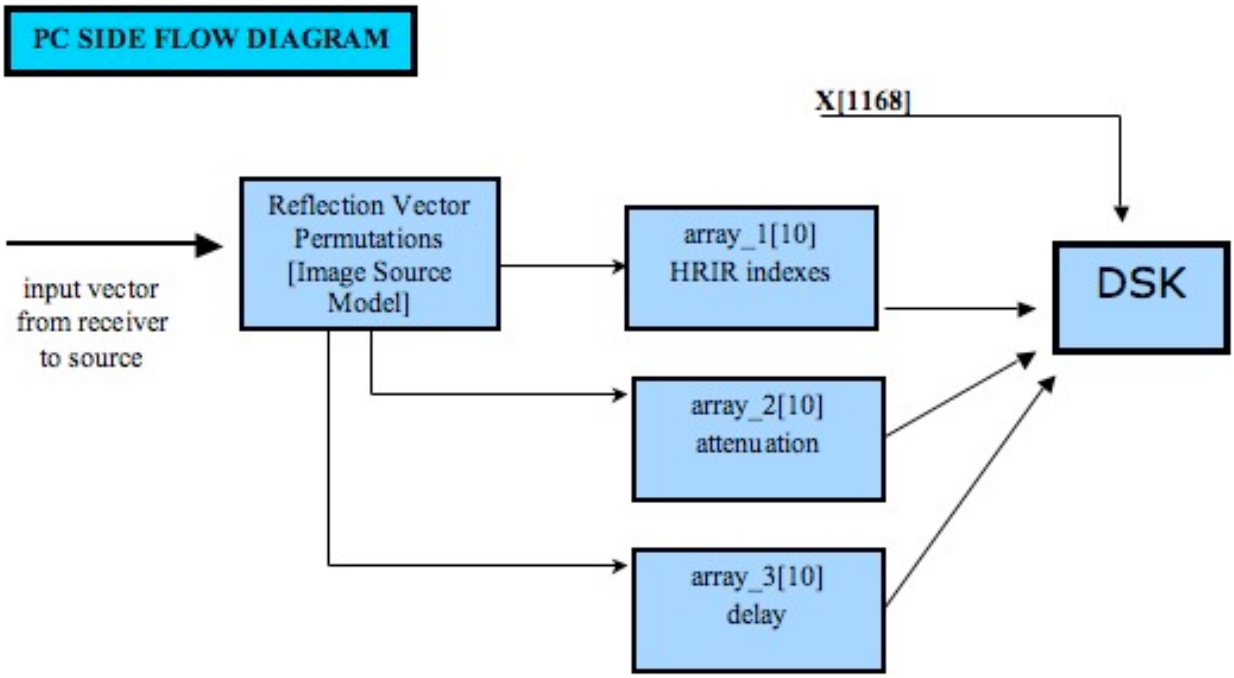
The intended goal of this project was to model the reception of sound in a small rectangular room as a user changes head orientation and position with respect to the location of the sound source. The proposed implementation would include filtering a monaural sound source with HRTFs (Head Related Transfer Functions) to reproduce the reception of sound by an individual after it is transformed by their specific anthropometry. Head Related Impulse Responses, which are used to obtain the HRTFs, vary somewhat from person to person but due to memory constraints we did not individualize the HRIRs by user.

Additionally, we had planned to perform room modeling. Each sound source creates a series of early reflections that have to be accounted for separately since they are perceptually distinguishable from one other. These can be generated by using the Image Source Model

which makes simplifying assumptions about the room to derive new receiver-to-source vectors. These echoes are filtered individually with separate HRTFs to account for different angles, delays, and attenuation. Thus, multiple head related transfer functions are necessary to model this phenomenon for a single sound source depending on the number of discernable early reflections in the desired environment. The last aural cue we had planned to include in our model is the late reverberation which can be generated using the Schroeder Late Reverberation Model.

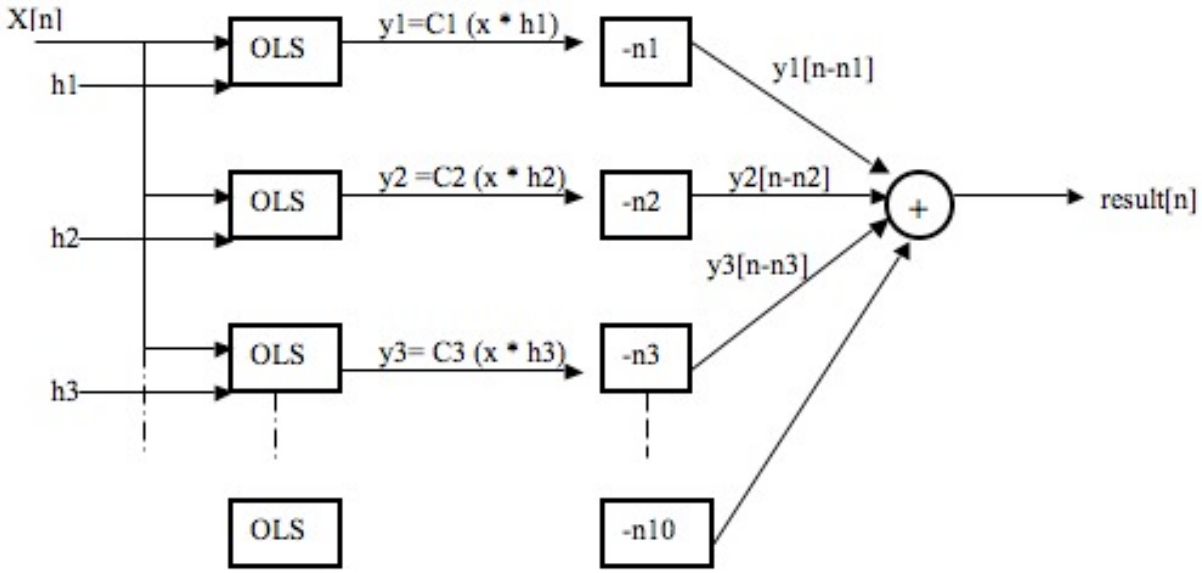
The following graphs represent how we intended to implement our solution. The PC would calculate the new position vectors for each of the early reflections in addition to their respective scale factors and sample delays. Thus, the information being transferred for each reflection would include the azimuth and elevation index, attenuation, and delay. This information would be sent in three arrays contained within a packet struct- one for the HRIR index for each reflection, one for the attenuation, and one for the delays. A block of the signal would also be contained within the packet. While the DSK processed the signal, the PC would calculate the Schroeder Late Reverberations using the original block of input signal to be added to the processed signal returned by the DSK.

PC Implementation Part I



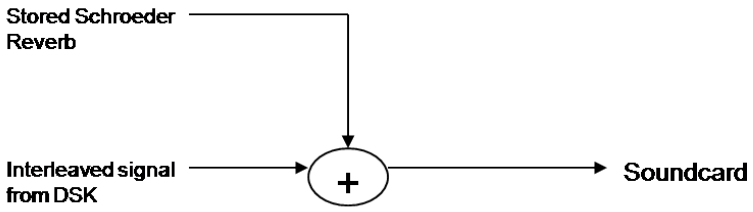
Once the DSK received this data, it would use the information associated with each reflection, convolve the corresponding HRIRs with the input signal, attenuate and delay each reflection by the appropriate factors, and sum the result.

DSK Implementation



The processing of the signal in the DSK would consist of taking the FFT of the signal as well as the FFT of the appropriate HRIRs for each reflection to obtain the corresponding HRTFs. Then the frequency response of the signal would be multiplied with the frequency responses of each of the reflections. Subsequently, the IFFT of the result would be taken- effectively convolving the HRIR with the input signal. This would be done for both the left and right side individually and then, the results would be interleaved before being sent to the PC.

PC Implementation Part



II

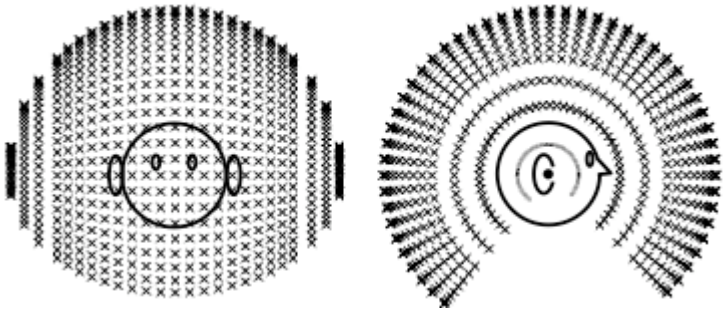
Once the PC receives the processed result from the DSK it would simply have to sum the Schroeder Late Reverberation Calculations and then output the signal through the soundcard.

As far as getting the system to operate in real-time the idea was that by chopping the sound signal into small process-able blocks we would be able to filter the signal a block at a time and output it a block at a time. Then we would simply have to optimize the code so that the output was fluid and uninterrupted. By implementing our system in this way we wouldn't have to limit the size of the overall signal. It would also make our code much more adaptable to receiving a streaming sound input which would be the ultimate goal in a virtual reality audio rendering system, although not within the scope of our project.

Databases and Algorithms

The UC Davis CIPIC HRTF Database

The database of head related transfer functions was taken from the UC DAVIS Center for Image Processing and Integrated Computing. The portion of the database which was made available on public domain includes measurements taken from 45 voluntary subjects and includes 250,000 data points for each subject. The data was generated by situating subjects at the center of a hoop of 1m radius where speakers were placed along the hoop and microphones in the subject's right and left ear. An impulse was, then, sent through the speakers and the microphones recorded an impulse response of length 200 for each ear at 44.1 kHz sampling rate and 16-bit resolution. The raw impulse responses were subsequently filtered using a hanning window to remove any reflections that may have been captured by the head related impulse response. This was performed for 25 separate azimuths ranging from -90 to 90 degrees and 50 elevations ranging from 45 to 130.625 degrees. The diagram below demonstrates the distribution of data points along the 1 meter hoop which were taken at irregular intervals for uniform sampling. (CIPIC Interface Database)



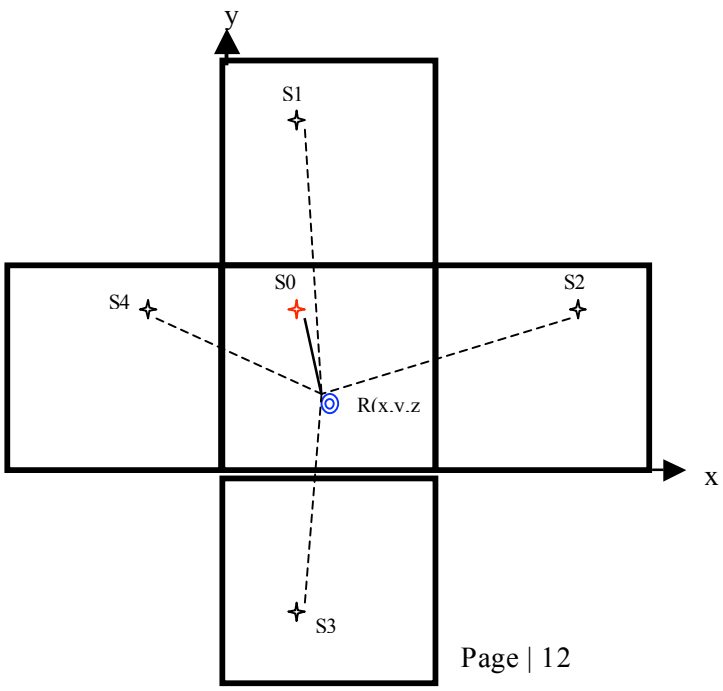
For the project, we chose to use the data from only one subject due to memory constraints on the DSK. Each database of 250,000 floats translated into ~1 Megabyte of memory. We assumed

that the difference in measurements between two separate subjects would be close to negligible to the human ear. Had time permitted, we would have experimented to see if the anthropometry made any difference. The types of measurements taken on each subject are thoroughly documented on their website.

Image Source Model

Traditionally, the derivation of room reflections in most room models require a series of complex computations in which each reflection has to be traced using trigonometric calculations. The model that we had intended to employ was the image source model, developed by Jont B. Allen and David A. Berkley at Bell Laboratories. The assumptions behind the model are that we are dealing with a room of rectangular geometry and that the walls are relatively rigid. This facilitates the calculation of new position vectors from the replicated sources to the receiver. In addition, Allen and Berkley claim that the solution derived from the image source model approach the exact solution derived from the wave equation as the walls become rigid.

Given a source and receiver vector in a three-dimensional cube, the first degree reflections can be calculated by mirroring the source over each wall to create a phantom source. The properties of these phantom sources are then calculated as if they were independent sound sources directed at the receiver.



Original Source Vector: S = < xs , ys , zs >
Receiver Vector: R = < xr , yr , zr >
Receiver-Source Vector: S-R = <xs-xr , ys-yr , zs-zr>

The subsequent reflections become the eight permutations of <xs ± xr, ys ± yr, zs ± zr>. And then, using arctan() you can derive the azimuth and elevation angles between the receiver and the sound source. (Allen and Berkley, 945)

PC Side Implementation

Our original plans called for the PC doing a lot of work. Among the PC's responsibilities were reflection vector calculation (i.e. finding the image source locations), calculating the Schroeder reverberation and merging it with the processed sound blocks returned by the DSK, as well as running the graphic user interface (GUI) and any other user input.

What we had originally planned on creating was a program which could take at least one input sound source and, in real time (or with at least minimal latency), process it using HRTFs, reflections, and reverberations to give it a directional quality. The idea behind the whole process was to be able to use the directional sound to give a user the feeling of a virtual environment using sound. With this goal in mind the project had to have several key elements to it: It must be able to change the sound's direction in real time, which implies moving either the source position or receiver position while the sound is playing and being processed. The program must also be able to use different room types and reverberation parameters to make the virtual environment more realistic. To do all of these things the GUI must keep track of receiver position, source

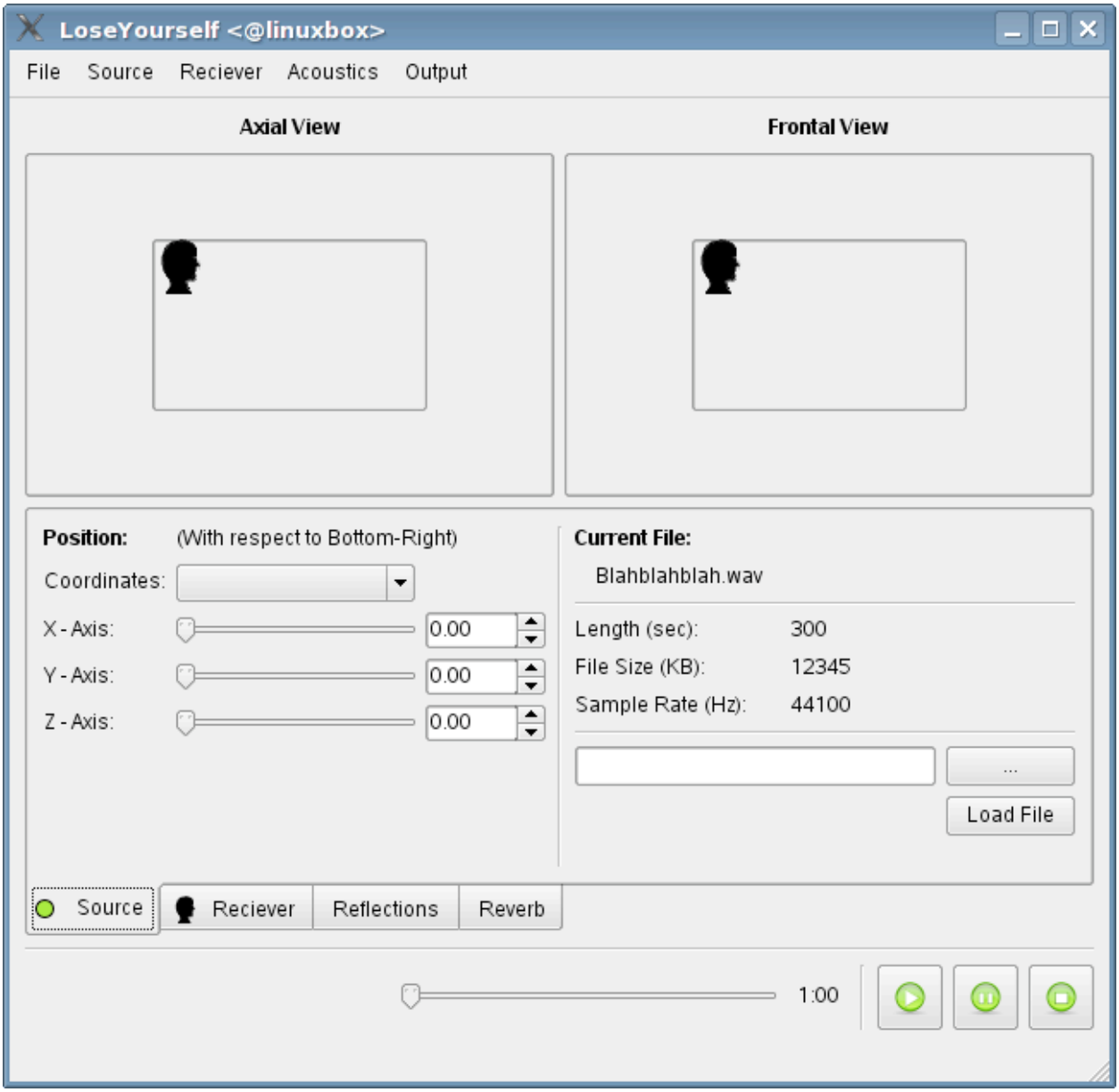
position, room dimensions and characteristics, reflection options, and reverberation parameters. It must also read in sound files, output the processed sound, and transfer and receive blocks of data to and from the DSK. All of this must be done asynchronously since the user can input new data and/or configurations at any time.

In order to handle all of these events we decided to use QT 4.3 to write our GUI. It uses a system of signals and slots to handle asynchronous events and it has many features which helped us out a lot, including integrated TCP networking, pre-made widgets (e.g. sliders, tabs, and input boxes), and very good documentation.

Our explanation of the PC side code will start with a description of the outer layer of the GUI, what the user would see and interact with. We will describe each feature and it's intended purpose as well as it's final state at the end of the project.

Below is a screenshot of the Source tab in the GUI. It contains sliders and boxes to change the position of the source. The user would have the choice of what coordinate system they wanted to work in using the drop down box above the sliders. This would let them work in either Cartesian (x, y, z) or Interaural-Polar (Phi, Theta, R) coordinates. The Cartesian coordinates would be referenced to the bottom right corner of the room, while the Interaural-Polar coordinates are referenced by the Receiver's position. Unfortunately, at the moment, the sliders only work to change the Cartesian coordinates of the Source. The boxes are not enabled and will not affect the Source position. To the right of the position coordinates is the source file information section. This section lists general information for the loaded sound file. This information is just set to default values since, currently, the file name and information must be

hard-coded.



Next is the Receiver tab. It is very similar to the Source tab in that it contains the same position inputs. It is slightly different though because the polar coordinate input is simple spherical coordinates with the bottom-right of the room as center. Again, the sliders are currently the only method of inputting a Receiver position.

To the right of the sliders and boxes is a section for inputting HRTF databases. It was planned to allow the user to input multiple databases and choose which sounded best to them. Unfortunately, this feature was not finished and the database file to load must be hard-coded. In the Reflections tab the user is able to choose how many reflections of each order they would like to be computed. The order number means the number of reflections before reaching the receiver. So first order is one bounce, second is two bounces, etc. Unfortunately, we were only able to calculate the first order reflections due to the complexity of the calculations along with time constraints. So while the option to change the number of reflections of each order is included in the GUI, the only one which has an effect is the first order.

Finally, the Reverb tab is very simple and just contains an on/off button for the Schroeder reverberation. This tab doesn't currently do anything since there isn't any reverb being used anyway. This tab will eventually have reverberation options on it as well, such as wet/dry setting and reverberation time.

Above all the tabs are two boxes which were going to allow the user to drag and drop the source and receiver positions as well as visualize the current positions. The receiver and source icons (seen on both Receiver and Source tabs) would be movable and would automatically change the position of either one when the user dragged the icon to a new position. Also, the icons would move and update if the user changed the position with the sliders or boxes. Since these features were mostly just eye-candy, they were not developed further than taking up space in the GUI window.

Below the tabs are several items: a disabled slider, a time, and play, pause, and stop buttons. The slider, along with the time stamp, was meant to show the current position of the sound being played, however, neither have yet been attached to anything. The play, pause, and

stop buttons were set up to do just as they sound. However, since there was no output signals from the DSK, this button was only tested on hard-coded sounds.

There are several action tabs at the top of the window (e.g. File, Source, Acoustics, etc.); while these were originally planned to have more options than the tabs had room for, they were never implemented due to time constraints.

PRE-DSK

Now that the shell of the GUI has been covered we will delve more into the structure and function of the code. To give the explanation more direction, first we will describe the flow of data. Also, to make the description easier and less wordy, we will use the assumption that the user only controls the source and receiver position for now.

When the program is started the first thing that is done is the GUI loads the HRIR database and starts a TCP server which listens for the DSK client. When the DSK connects to the TCP server a socket is opened and the HRIR database is sent to the DSK for storage. After the HRIR is sent the GUI initializes the SDL libraries which includes initializing the sound card for output. This involves sending the SDL libraries the output specifications which are currently hard-coded to 44.1 kHz sample rate, 2 channels (stereo), and a 1168 sample buffer. The SDL libraries will open the sound card for writing and pause playback. At this point the GUI, using the SDL libraries reads a wave file into memory, using the class AudioFile as a storage and retrieval medium.

The GUI is now ready to start sending and receiving packets of data to and from the DSK. When the GUI receives a command from the DSK to send a packet of data it gathers together all the needed data and puts it into a pre-made packet structure. The structure contains

many different bits of data: a sequence number, an end of signal flag, an exit flag, 1024 samples of signal data, and several variables describing the positions of any image sources. The sequence number will be used later once the result packet has been sent back from the DSK and the end of signal and exit flags have obvious functions. The 1168 samples of signal data are arranged so that the first 199 overlap the set of samples from the previous packet.

The set of variables which are used to describe the image sources are interaural-polar coordinates, attenuation coefficient, and sample offset. So the packet sent contains one variable with the number of image sources (up to 9 plus the original source) as well as four arrays: one containing the azimuth variables of all sources (image and original), another containing elevation, another attenuation, and lastly offset. Azimuth, obviously, means the position of that source along the azimuth arc. Elevation is the same as azimuth, just in the up-down arc. Attenuation means the amount to multiply the signal by for this particular source in order to account for wall absorption and inverse square law. The final parameter, offset, is the number of samples to offset this source by from the original source to account for extra distance traveled. Both image sources and the original source are affected by attenuation. The reason the original source is affected by attenuation is because the attenuation due to the inverse square law gives the sound distance. Even though both image sources and the original source are affected by distance attenuation, only image sources are affected by offset since, with only one original source, all offset is relative to that original source.

The vector containing all of the information describing the image sources is calculated by the GUI anytime the user changes either the room configuration, the source position, or the receiver position. This way the vector is always up-to-date and a packet can be sent at any time without recalculating the image source vectors. While we were not able to test the overall speed

of our send and receive loop, we assumed that it would be fast enough so that the user would not notice any latency in the movement of either source or receiver.

Reflection Calculation

The reflections, or image sources, are calculated using a method which essentially reflects the room and source across one of the walls and uses the new position of the source as an "image source." The diagram below shows how this geometric reflection of the room and source leads to image sources. The number of reflections is an exponential function, so for any particular order of reflections there are up to (6^N) image sources in a three dimensional rectangular room, where N is the order of reflections. With the number of image sources growing so rapidly, not many orders need to be calculated to get an overwhelming number of reflections. This is why we limited our reflections to four orders (although this is probably more than enough).

Reverberation Calculation

For each packet of signal data that is sent to the DSK a reverberation signal is calculated using the sent signal. This reverberation signal is stored for retrieval after the corresponding result signal is sent back from the DSK.

The reverberation signal is calculated using a Schroeder reverberator. The code which we used to implement the reverberator is called Freeverb (ccrma.stanford.edu). It uses eight comb filters and four all-pass filters on each channel. It allows the user to change several different parameters: room size, damping, wet/dry, width, and mode. The documentation is fairly poor, so we had to guess how many of these parameters affect the final output. Wet/dry is pretty common

and changes the amount of reverberation signal ("wet") and unchanged signal ("dry") which are returned. The damping parameter determines how much of the higher frequencies the "room" absorbs. Room size and width seem to be related, but there is no exact definition in the documentation as to what they do or how they change the output. Our best guess is they make the room sound larger by increasing the delays. Finally, mode determines whether the reverberation should be held. In other words if the signal is being sent in blocks, like we are, should the reverberation be held over from block to block.

POST-DSK

After both the signal block and image source vector have been sent to the DSK and the corresponding reverberation signal is calculated and stored, the GUI must wait for the DSK to return a result block of signal. When a result packet is sent by the DSK, it will contain several things: a sequence number, and two arrays of 969 samples of signal. The sequence number is just a check to make sure the packets are arriving in order and also to help align the result data with the reverberation data. The two arrays contain the left and right samples of the result signal. These are the outputs from the FFTs. These signals emulate what the left and right ear should be hearing, taking into account the HRTFs and reflections. The last part in making the sound more realistic is adding the stored reverberation. Once this reverberation is directly added, the two signals, left and right, are interleaved in preparation for output to the sound card.

Sound Card Output

While all the other calculations are going on the sound card must be continuously filled with valid samples, otherwise the user will hear blips and glitches where random data is sent out.

The way the SDL sound processing works is, when a device is opened a callback function must be given. Whenever the sound card needs new samples to put in it's buffer, it calls the specified callback function. We wrote this function to then call a function in our Soundcard class which holds a circular buffer of current output data. Every time the sound card calls the callback function a certain number of samples are written to the sound card's buffer and the head of the circular buffer is moved forward. Also, anytime a packet is received from the DSK, after being added with the reverb, it should be added to the circular buffer. There are several methods which track both the size of the buffer and the receptivity of the DSK, making sure that new packets go out as soon as possible. However, we were unable to test this system due to problems integrating the DSK and the PC as well as time constraints. Even without test, though, we believe that the DSK was processing fast enough to fill the buffer before there was any underrun.

What We Implemented

Sound Signal

We implemented our system to work with a sound signal that satisfies the following characteristics:

- 44.1 kHz
- Monaural
- Binary File

The last requirement has to do with how we implemented the readDBsFromFile() function on the PC. If one wanted to read files saved in a different format one would have to edit the function accordingly.

Memory Allocation

The HRIR Databases for each user are 1MB per side. This adds up to a total of 2MB when you take both the left and the right. Since we only have 256KB of internal memory we had no choice but to place them in external memory.

In our original implementation design, only the databases would have been stored in external memory. Everything else, including the signal blocks, would have been stored in internal memory. This way it would be more accessible and require less access time therefore making our code run faster. However, in our final implementation we ended up sending the input signal all at once to the DSK, instead of chopping it up into smaller blocks and sending the signal over one block at a time. This meant that the entire signal had to be store on the DSK. Given that the signal we chose to process required ~2.5MB of memory we had to store it in external memory.

Algorithm Implementation

The first step in our implementation was to read in each of the databases for the right and left ear and the input signal from file into three float arrays onto the PC. Then, using a socket network connection with the DSK as the client and the PC serving as the host, the three signals would be sent to the DSK to be stored in external memory. On the DSK side we had to allocate memory for the databases and the signal.

Once the DSK allocates memory and receives the databases and the input signal it calls the overlapSave function to process the information. Based on the size of our input, the size of

the input block that overlapSave processes with each individual call (1168*sizeof(float)), and taking into consideration the fact that every time overlapSave is called the block of input signal processed overlaps the block processed in the previous call by 199 samples, we determined that, for an input signal of the size being used, we needed to call overlapSave 625 times to ensure that the signal would be processed in its entirety.

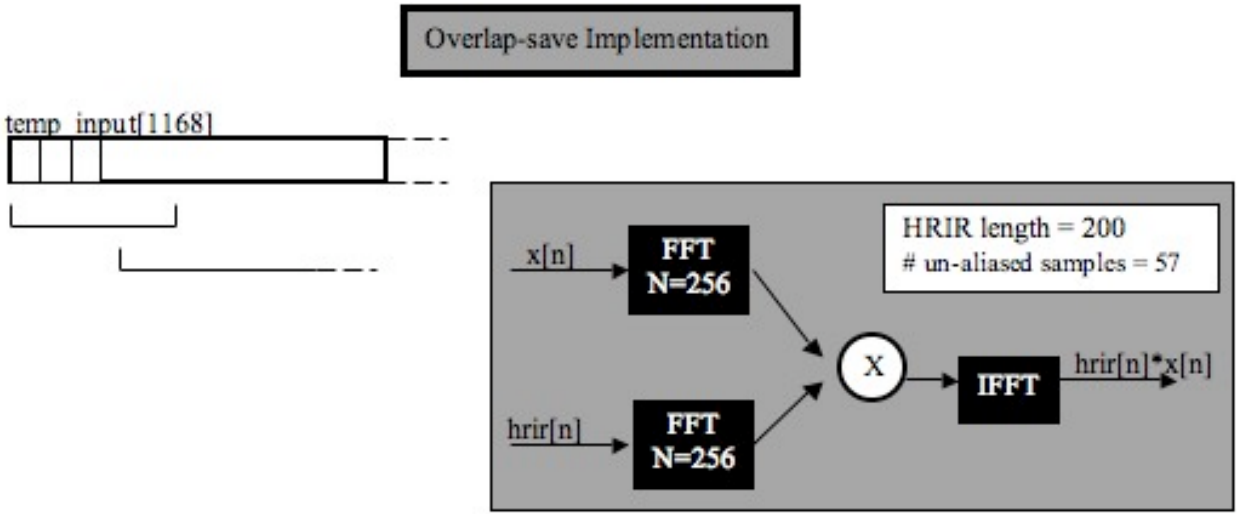
In order for overlapSave to be called, we needed to calculate the reflection index that it takes as a parameter. In each iteration, a new HRIR is selected based upon the given azimuth and elevation. This reflection index is fed to the overlap save function to select the appropriate impulse response whose first element is indexed by the following equation:

Index = 200*azimuth_index + 5000*elevation_index

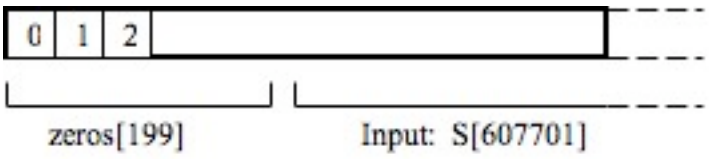
Originally, every input signal block was supposed to be sent to the DSK with accompanying arrays containing the azimuth and elevation information for each reflection as well as for the original signal (along with other information). We were not able to achieve the full functionality we desired on the PC side code. As a result, instead of obtaining azimuth and elevation information from the PC, we simply hard coded their values. One of the ways we did this was using two nested for loops in such a way that every 25 samples the azimuth would increase by one, starting at 0 and ending at 24. In this way we would cover the entire azimuth range. The result was that the processed signal sounded as if the speaker started speaking on your left side and then walked around you while he talked until he ended up speaking on your right side.

This HRIR is read into a local static array, interleaved with zeros, and transformed using a 256 point FFT using the Cache –Optimized Mixed Radix FFT function provided in the c67x

library. The 1168 points of the full input array are, then, read into a static array and taken 256 points at a time to be interleaved with zeros and Fourier transformed. The HRTF and the frequency response of the 256 point input blocks are multiplied and inverse FFTed with the Cache-Optimized IFFT. Taking the last 57 samples of the output gives us the un-aliased convolution result. Each call to overlap save produces 969 samples of filtered signal.



The above diagram illustrates our implementation of overlap-save in which the algorithm is broken down to process 1168 sample blocks at a time. The array below shows how we accounted for the first aliased 199 samples in the input array-by zero-padding once.



At the end of each call to overlap-save, the resulting segment of filtered input (N = 969) is sent back to the PC where it is written to a file. Thus, 1250 files in total are written to file and

comprise the output from both the right and left signals. We, then, wrote a Matlab function to patch together the files into a 607701 x 2 matrix and played with soundsc to verify that the filtering was successful.

FFT/IFFT

The Fast Fourier Transform and Inverse Fast Fourier Transform we chose were the cache-optimized mixed radix single precision floating point functions provided by the TI c67x library. Our initial reasons for choosing these two algorithms were the benchmarks provided on the TI website in which the Cache Optimized algorithms were projected to out-perform the Decimation in Frequency algorithm for N=256. The number of cycles for each single precision floating point fft are listed below for N=256:

-SP Complex DIF FFT (radix4)

(14*n/4 + 23)*log4(n) + 20, cycles = 3696

-SP Out-of-Place Cache-optimized mixed radix FFT w/ digit reversal

3*ceil(log4(N)-1)*N + 21 * ceil(log4(N)-1) + 2*N + 44, cycles=2923

-SP Cache-optimized mixed radix Inverse FFT

3*ceil(log4(N)-1)*N + 21 * ceil(log4(N)-1) + 2*N + 44, cycles=2923

We also based our decision off of the profiled results from Fall 2007, Group 7's midterm presentation in which they had compared the number of cycles needed to execute the cache-optimized FFT to that of the decimation in frequency algorithm.

Optimization

Because we didn't obtain the functionality we needed on the PC side, we were unable to do much optimizing beyond that which pertains to data transfers. There are two main instances that occur in our code were data transfers between external and internal memory. Each instance is called numerous times during the execution of our code. The first instance consists of transferring the block of signal to be processed from external, where all ~2.5MB of the signal are stored, to internal. We estimated the required transfer time for 1168 sample block.

CPU Transfers (a.k.a for loops) : 7.125 cycles/word * 1168 words = 8,322 cycles

EDMA Transfers (using dmaCopyBlk) : 2.25 cycles/word * 1168 words = 2,628 cycles

The actual transfer times calculated using Function Profiling were as follows:

CPU Transfers : 9, 007 cycles

EDMA Transfers : 3, 127 cycles

So in this case, for the purpose of optimization, we employed dmaCopyBlk to transfer our data from external to internal.

The other instance involves accessing the appropriate impulses from the HRIR databases:

We estimated the following times:

CPU Transfers:

worst case (in internal then external) = 7.125 cycles/word * 200 words = 1,425 cycles

best case (found it in internal) = 1.5 cycles/word * 200 words = 300 cycles

EDMA Transfers : 2.25 cycles/word * 200 words = 450 cycles + 30 cycle overhead = 480 cycles

In practice we obtained the following numbers:

CPU Transfers: 632 cycles

EDMA Transfers: 653 cycles

We believe that this is partly due to the fact that it when the position doesn't change (i.e. azimuth and elevation don't change) then the CPU transfer approaches the best case scenario because it keeps accessing the same information. In this case we decided to go with the CPU transfer because it gave us the best time (even if only marginally) and because our implementation is such that the transfers are closer to a best case scenario. If the implementation were to be changed (like say for example to finish implementing our proposed project) this particular decision would need to be re-evaluated for the sake of optimization.

Finally it should be noted that our code was run at an optimization level 3 (as in 18-551Lab 3).

The Problems We Encountered

Over the course of our project there were a few notable roadblocks that impeded our progress and are noteworthy for future 18-551 project groups. The first problem we had encountered in implementing our algorithms on the DSK was how to properly use the Cache-Optimized Mixed Radix FFT and IFFT.

FFT and IFFT Implementation Problems

In the given user guides, the buffers referenced by the input pointer *x and the result pointer *y must be double word aligned. Unfortunately, when we researched the term, we misunderstood the definition and did not understand how this requirement would affect our code. Double worded means that the complex time data, x, and twiddle factors, w, have to be aligned on double-word (8 byte) boundaries. Originally when we were writing our code we declared the following variables:

```
static float *hrir_database_R; //4 bytes
static float *hrir_database_L; // 4 bytes
```

before declaring the variables that would be used in the FFT:

```
static float hrtf[512];//multiple of 8 bytes
static float intermediateResult[512]; //multiple of 8 bytes
static float w[512]; //multiple of 8 bytes
static float block[512]; //multiple of 8 multiple
```

By pure coincidence the pointers, which were declared first, totaled 8 bytes and therefore the double word alignment was maintained. As a result we were under the impression that we had gotten the both the FFT and the IFFT working. Later, we modified our implementation to add another float pointer as follows:

```
static float *hrir_database_R; //4 bytes
static float *hrir_database_L; // 4 bytes
static float *s;//4 bytes


static float hrtf[512];//multiple of 8 bytes
static float intermediateResult[512]; //multiple of 8 bytes
static float w[512]; //multiple of 8 bytes
static float block[512]; //multiple of 8 multiple
```

This new float was 4 bytes and consequently shifted the buffers below it by four bytes. Since we did not explicitly cast our parameters appropriately, our resulting fast Fourier transform outputs

were incorrect. Because we did not understand what double word alignment meant, we wasted a lot of time trying to determine why the FFT and IFFT provided the correct result one moment and, then, entirely wrong answers the next. We mistakenly came to believe that the problem had something to do with memory because the FFT suddenly stopped working when we added a float pointer and allocated memory in external. Eventually, with the help of one of the TA's (Rohit Patnaik), we were able to determine the actual problem- that our inputs were not double word aligned as the documentation stated. The quick solution that we implemented was to simply move the pointer declaration beneath the buffer declarations. Since the buffers all had sizes multiples of 8 bytes they were naturally double word aligned and the FFT worked properly. However, it may not always be the case that your variables fall on a double-word aligned boundary. With Rohit's help we learned that the proper way to make sure that your buffers are double word aligned would be to follow the directions on page 139 of TMS320C6000 Optimizing Compiler User's Guide (Rev. N) (http://www-s.ti.com/sc/techlit/spru187n):

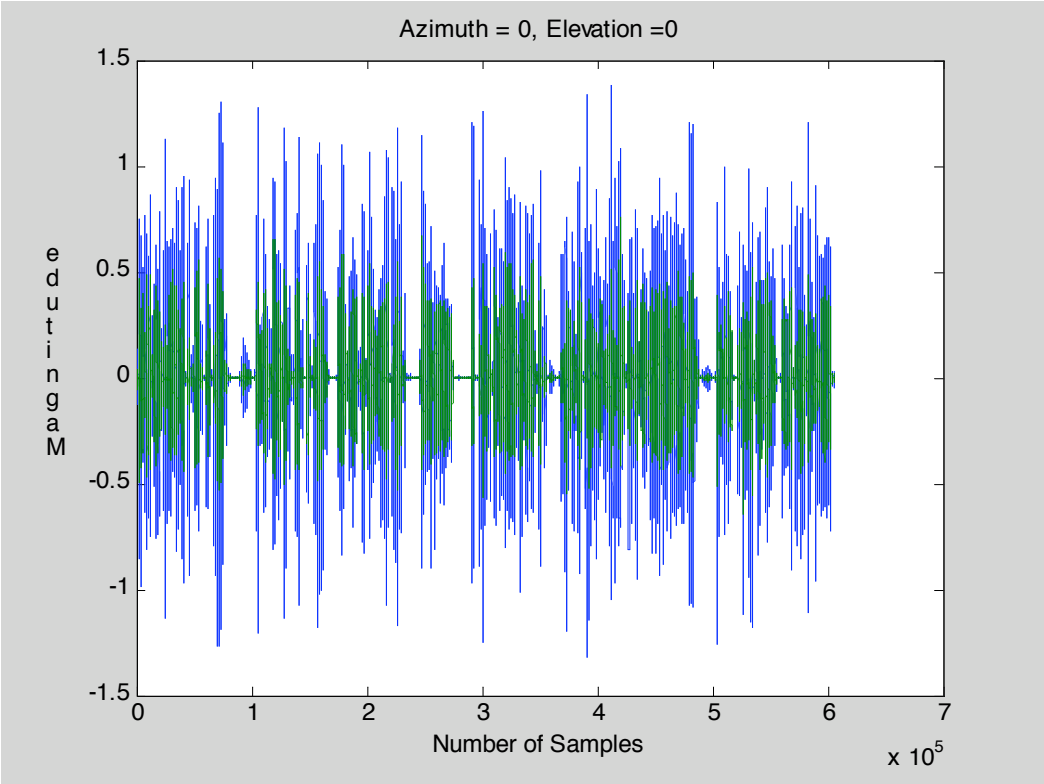
```
#pragma DATA_ALIGN (array_x, 8);
float array_x[256];
```

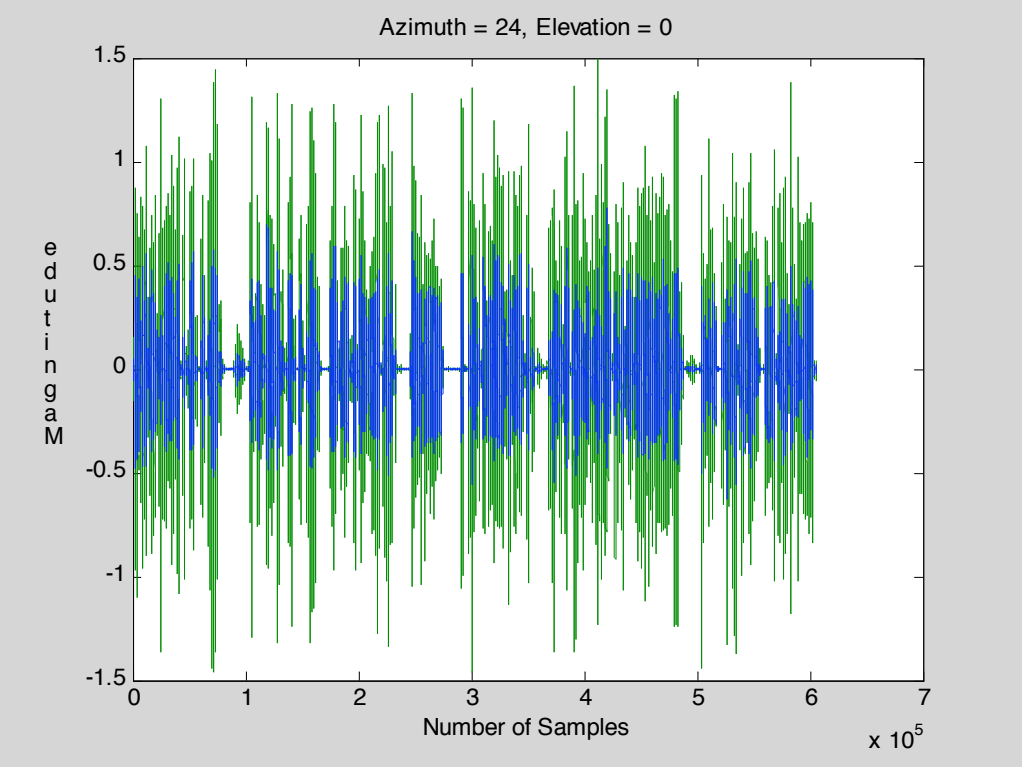
This will ensure that array_x is aligned on an 8-byte boundary, regardless of when the array is declared with respect to the other variables.

Results

We implemented our system using various variations in azimuth and elevation. For the purposes of observing the basic behavior of our code we processed the signal using Azimuth = 0 and Elevation = 0 and then using Azimuth = 24 and Elevation = 0. When the azimuth equals zero this corresponds with the sound source being on your left size and when it equals 25 it

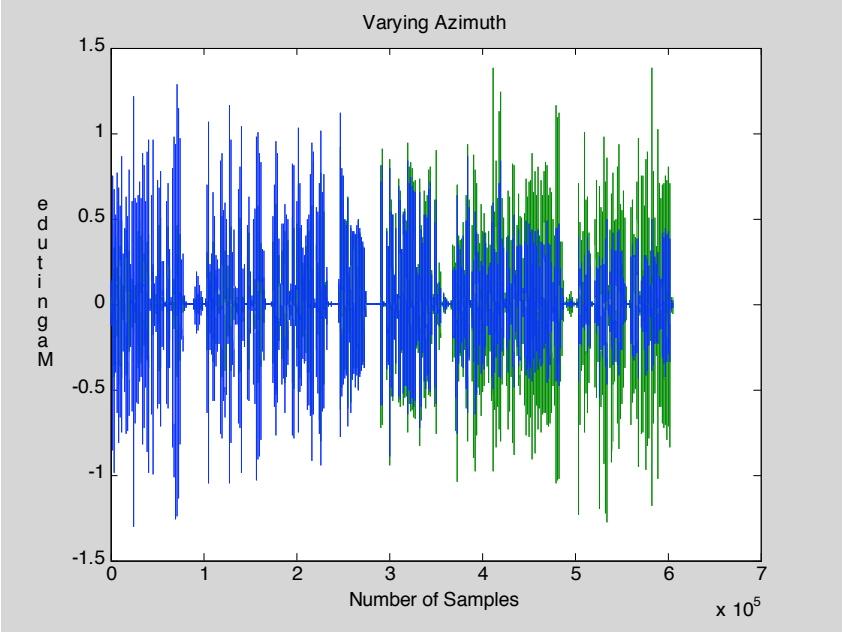
corresponds with the sound source being on your right side. The following plots consist of the magnitude vs. sample number for each of the two mentioned implementations.

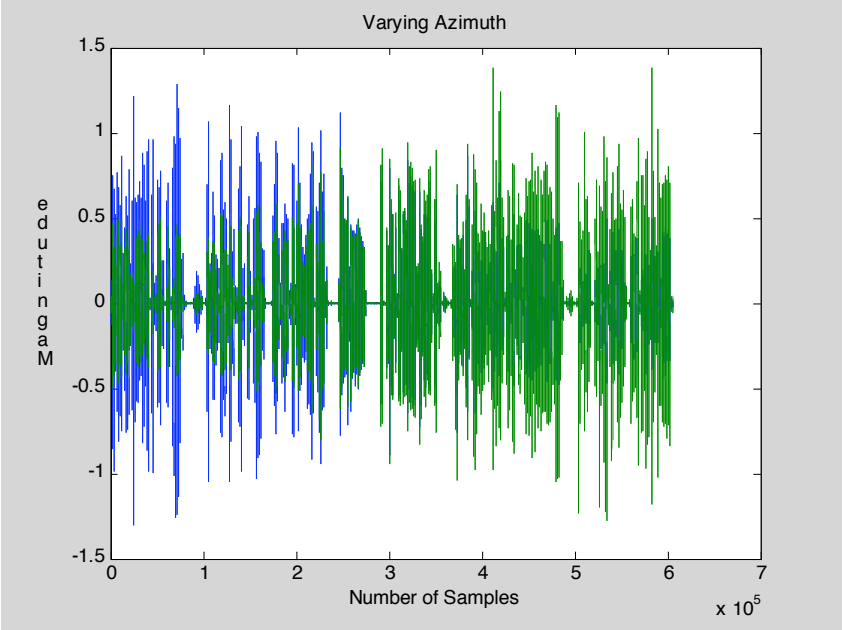




The blue signal is the left side signal and the green signal is the right side signal. The left hand signal is much greater in magnitude when the azimuth is zero. This makes sense since this is when the sound source is on your left side which would mean that the signal would sound much louder in that ear. When the azimuth is 24 the situation is reversed.

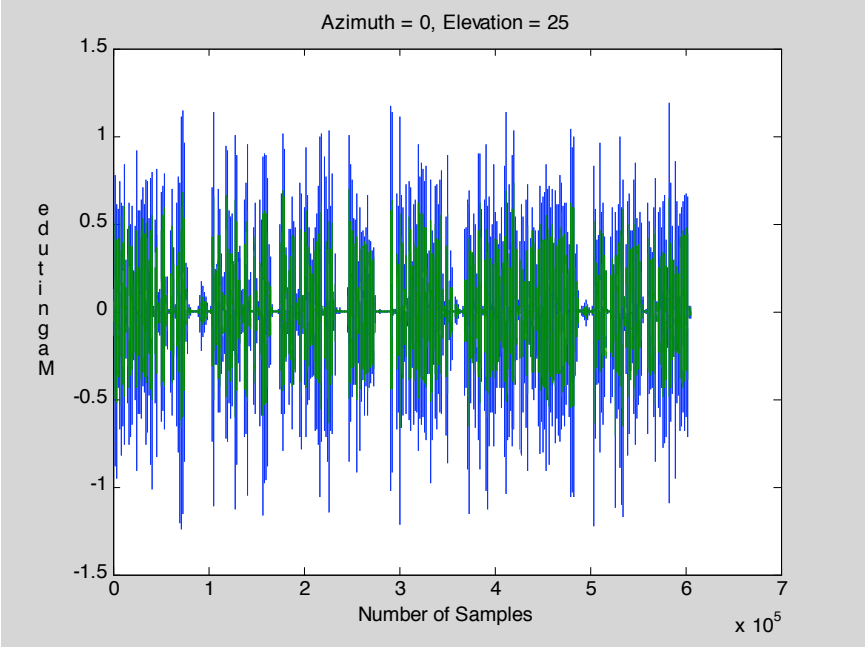
In the following figures we've plotted the case where the azimuth varies from 0 to 24 while the elevation remains constant. Again you can see that initially, when the azimuth is zero, the left side signal (the blue one) has greater magnitude but as the sound moves towards the right side (an azimuth of 24) the right side (the green one) starts to gain in magnitude as the left side decreases.

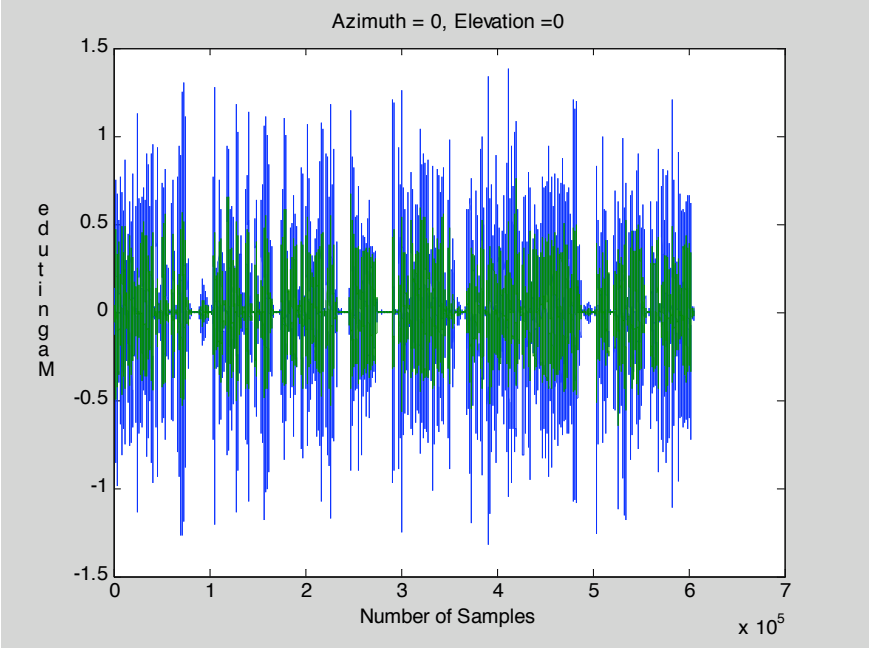




The two plots represent the same varying signal. The only difference is that in the top figure the left side signal is in front of the right side signal and in the second figure it is behind the right side signal.

As far as variations in elevation are concerned, plots for signals processed using different elevations but with a fixed azimuth present a slight variation. But the changes were so small as to be barely noticeable.





The changes in elevation were even harder to hear than they were to see in the plot. We were unable to distinguish any difference in elevation from listening to the processed signal let alone determine the origin of the signal, elevation-wise.

Conclusion

Our system implemented the core HRTF filtering very successfully, but we failed to deliver when it came to the novelty part of our project. Changes in azimuth were audibly distinguishable but changes in head elevation are go unnoticed. We have yet to determine the cause of this or how to incorporate other spatial cues to suggest a sound source location. According to the research that we did prior to our implementation, the Schroeder model and the Image Source model would have served as sufficiently accurate representations of reality for the small room environment we had intended to simulate. Future 18-551 groups can do a multitude of other variations of this project in addition to our intended implementation of room acoustics. Dynamic control and real-time processing are endeavors worth pursuing as possible projects.

References

Allen, Jont, and David Berkley. "Image method for efficiently simulating small-room acoustics." The Journal of the Acoustical Society of America 65.4 (1979): 943-950.

"CIPIC Interface Laboratory." CIPIC Interface Laboratory. 10 Dec. 2007 <http://interface.cipic.ucdavis.edu/index.htm>.

V. R. Algazi, R. O. Duda, D. M. Thompson and C. Avendano, "The CIPIC HRTF Database," Proc. 2001 IEEE Workshop on Applications of Signal Processing to Audio and Electroacoustics, pp. 99-102, Mohonk Mountain House, New Paltz, NY, Oct. 21-24, 2001.

"TMS320C67x DSPs." Focus Home Page. 10 Dec. 2007 <http://focus.ti.com/dsp/docs/dspplatformscontentaut.tsp?familyId=327&sectionId=2&tabId=499#fft>.

Trolltech, QT 4.3, <http://trolltech.com/products/qt>

Freeverb, Written by Jezar Wakefield at Dreampoint Design and Engineering. Note: We were unable to find a website for Dreampoint Design and Engineering, and there were numerous copies and rewrites of the original Freeverb online. So here is the best site we could find which offers the source code and a simple explanation of the original code: http://ccrma.stanford.edu/~nando/clm/freeverb/