18-551 – Fall 2008 – Group 3

2-Point 5.1

Headphones Never Sounded This Good

Final Report

Dan Passmore (dpassmor@andrew.cmu.edu)
Vitaly Cherednichenko (vitalyc@cmu.edu)
Yiran Liu (yiranl@andrew.cmu.edu)

Table of Contents

Introduction

The Problem:
There remains a disparity between the world of high-end digital surround sound entertainment and the common pair of headphones. When the average college student is in the mood to enjoy the latest, greatest cinematic production in the fullness of its surround sound glory, the worst thing that can happen is discovering that the roommates are asleep, preventing the use of a fancy surround sound system. Having to resort to using headphones in place of a real surround sound system usually ends up ruining the entire experience. If there was an effective way to transform three dimensional sound sources to standard stereo headphones in real time without losing the effect and power of a real surround sound system, then every common college student would be free to enjoy their favorite movies at any absurd hour of the night using nothing more than an ordinary set of headphones, without waking up the neighbors or losing the priceless pleasure of a true surround sound experience.

The Solution:
We proposed to design and construct a DSP system which will model the human head and convert a five channel surround sound signal into a standard two channel stereo output in real time without losing the acoustic qualities of a genuine surround sound experience. We proposed to implement our solution on the DSK using pre-measured Head Related Transfer Functions which will be optimized to run in real time on the DSK with minimal latency. We will be sending uncompressed 5 channel audio data to the DSK (read from a file on the computer) and the DSK will transform the music into a real-time stereo simulation of the surround sound environment and send it to the headphones through its integrated headphone jack.

What We Have Done:

We have taken a 5.1 channel sound file, uncompressed it, and split it up into 6 wave files using VLC. VLC also streamed this data over the internet (using HTTP over TCP) to our PC.

We have written a java process (We have dubbed PC Packager) on the PC that is responsible for receiving the VLC TCP stream, keeping track of how the user modifies the GUI, and sending the DSK the data to process, or the filters that need to be updated for the HRTF.

We have implemented a C code version of the Mixed Radix FFT algorithm on the DSK (by using TI's code).

We have successfully shown how to take 6 real FFTs with only 3 FFTs through our own implementation.

We have implemented the overlap save method the DSK using the above two things.

We have Created a PC Player Process that receives filtered data from the DSK and simultaneously plays and stores that data in real time.

We have broken ground with UDP (We are the first to use it ever in 18551) by getting transfers to and from the DSK to work in the UDP protocol.

We have defined some very simple UDP protocols that are used in these transfers.
Prior Work:
There were two previous 18-551 projects that were similar to ours. “Lose Yourself: A Virtual Reality Audio

Immersion" by Group 10 of Fall 2007 and "Music to the Ears: Creating Multi-Dimensional Sound for a Virtual Environment" by Group 2 of Spring 2001 both dealt with implementing HRTF's to represent an audio environment.

References:

Previous Groups:
Spring 2001 – Group 2 Report
Fall 2007 – Group 10 Report

Database:
CIPIC Interface Laboratory Website: http://interface.cipic.ucdavis.edu/index.htm
Download Database Here: http://interface.cipic.ucdavis.edu/CIL_html/CIL_HRTF_database.htm

Textbooks:
Digital Signal Processing, Principles, Algorithms, and Applications - Proakis, Manolakis
(this was used for overlap save method and for checking up those FFT symmetry properties)

How We Differ:
There are several key differences between our project and the two similar projects conducted in the past. We will get all our data to process on 6 channels, we will be using an extremely high quality sampling rate of 48 kHz for all our calculations and implementation, we will be using UDP instead of TCP to achieve better network simplicity and less processing of packets. Finally, our project will be in real-time, a feat the previous two groups did not achieve.

What is New:
The application is not new (Dolby for example has something called Dolby Headphones that is very similar to what we are doing), but putting this technique all into one neat package that works well with VLC and is for personal use with existing media has not been done before.

Algorithm Overview

The basics

It turns out that DSP-wise, our project is fairly (dare we say it?) simple. We are basically just filtering 6 channels with a set of fixed filters (2 for each channel), and then adding the results into two output channels. There are clever, somewhat complicated ways of doing this filtering quickly - which we have done - but for the most part, the complicated part of this project is the real time aspect and getting the DSP done FAST! Here is the basic flow of our system:

DATA FLOW GRAPH

DVD Movie with 5.1 Audio or Dolby Digital Surround Sound (DVD or file)

↓

VLC:

Play just the Video Locally

Stream 6-channel Wave files over TCP to the PC-side packager

↓

PC-side packager:

De-interlaces audio channels and sends them as floats to the DSK over UDP

↓

DSK code:

Unpack UDP data and load it into memory

↓

Perform Filtering on audio

↓

Scale and Mix the 12 filtered channel results into two channels

↓

Pack Stereo data into UDP packets and send it back to the PC

↓

PC-side Wave assembler and Audio Codec:

PC receives streaming data from the DSK, interlaces it, and re-assembles it into a Wave file

↓

PC Plays reconstructed Wave audio stream through its sound card

↓

PC also Saves the output as a Wave File

The key point to recognize here is that there are 4 different processes communicating with each other in real time (bold/underlined in diagram above).

First, there is VLC, which is the source of our data. It plays any user selected video locally and is setup to stream 6 wav channels at 48 kHz each over the network using TCP. The reason VLC talks using TCP is because we could not easily configure it to do the same thing over UDP. (Not that it needs to be said but just to be clear, we obviously did not write the code for VLC. We simply configured it).

Second, there is the PC packager process, which receives and converts VLC data into data that is ready to process on the DSK. This data is stored and then sent via UDP to the DSK whenever the DSK sends a packet saying it is ready for more data to process. Additionally, the PC packager process contains the GUI and will send filter information to the DSK in real time using UDP whenever the GUI is updated by the user. These filter packets, which take priority over data, will take the place of data packets and cause momentary (very small) gaps in playback. For more information on the GUI and its interaction with the Packager, see the GUI section below.

Thirdly, there is the DSK process, which does the actual filtering and DSP related functions of our project with the data sent from the PC packager. The DSK sits in a loop where if it is not currently processing received data, it will send "I am ready" UDP packets back to the PC packager. When the DSK finishes processing any data packet, it sends the results to the PC player process using a UDP packet and also sends back another "I am ready" UDP packet back to the PC Packager. If the packet it receives is a filter packet, instead of running through the filtering process, it instead installs the filter in its memory and then sends another "i'm ready" packet.

Lastly, there is the PC player process which plays and saves the output from the DSK. It gives no

feedback to the DSK and only listens to what has been sent via UDP.

Broken down into DSP vs PC functions, we have the following:

What the DSK and PC are doing

What the DSK is doing (basically all the DSP stuff)	What the PC is doing (basically moving data around and GUI)

• Receiving Filter Coefficients from PC
• Receiving input audio data
  • 6 channels of 48kHz 32-bit floats
• Filtering and Mixing
  • 12 different filters (6 per ear)
  • 6 channels to 2 channels
• Send output data
	• Send Filter Coefficients to DSK
	• Reading the audio data from a file and sending it to the DSK
	• Receiving the processed data back from the DSK to play and store
	• GUI

Algorithm Details

Shooting for Maximum Efficiency

Overlap Save

Since our project is a filtering project, we have looked for ways to do this as efficiently as possible. A previous project found that the fastest (least cycles) way to do filtering using the DSK was using the overlap save technique, and therefore we chose this over other techniques. This algorithm is based on the idea that multiplying in frequency is the same as circularly convolving in time.

Because we are streaming our data to the DSK over time and we don't have all our data all at once, we have to take the FFT and IFFT of smaller chunks and then throw out the data corrupted in the circular convolution (Length of Filter minus 1 - see database and signals section). In reality, given that taking the FFT of something is an order of N x log N process and that we are throwing out a fixed number of output samples per chunk, there is an optimal chunk size that yields the highest ratio of [amount of data outputted / the number of cycles used to compute the samples].

Previous projects have performed these calculations to determine the ideal chunk size to use. With our project however, we didn't have much choice with the chunk size due to memory constraints (See Memory section). Considering chunk sizes have to be a power of 2 to do FFTs efficiently, we could either go with a chunk size of 256 or 512 samples (1024 wouldn't fit in internal memory, and 128 is smaller than our filter size). Since our filters are 218 samples long, these would yield 39 or 295 samples. We made the decision without testing that 512 was the way to go, because getting 39 samples for every 256 length FFT is almost certainly less efficient compared to getting 295 samples for every 512 length FFT.

The rest of the overlap add algorithm involves multiplying the FFT'd input data by the FFT of the filter (which we keep stored in internal memory) and then IFFTing the result. Since there are 6 channels per ear that need to be added up in time after the filtering, we saved some extra processing time and added the channels in frequency before taking the IFFT. This reduced the number of IFFTs we needed to perform from

12 down to 2.

Below is a diagram showing how overlap save works, taken from page 487 of our old DSP book. (Digital Signal Processing, Principles, Algorithms, and Applications - Proakis, Manolakis)

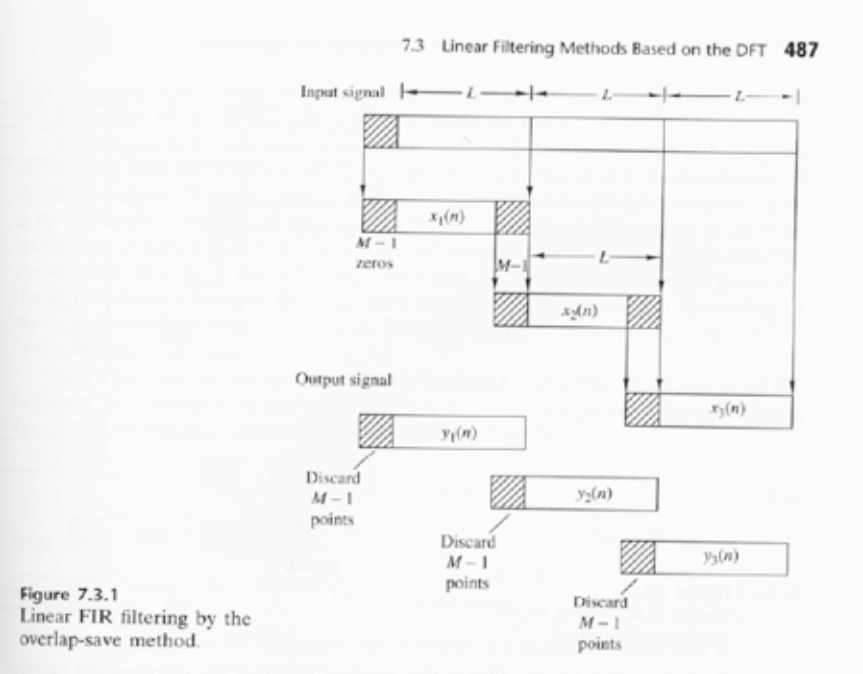


Figure 7.3.1
Linear FIR filtering by the overlap-save method.

FFTs

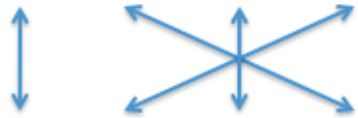
Obviously in order to perform the overlap save routine, we need to be able to perform FFTs efficiently. We can not use the FFT algorithm from lab 2 because that is a radix 4 implementation and 512, our chunk size, is not a power of 4. Because of this, we have used a C code version of a mixed radix algorithm provided on TI's website (We tried other versions that didn't work. See "didn't work" section).

To gain further efficiency out of our FFTs, we rely on some Fourier Transform symmetry. When it is desired to take FFT of two input signals and the input data is all real (which ours is), instead of taking two separate FFTs, we can make one of the inputs the imaginary part of the other input, and then by performing one FFT and using symmetry properties, we can separate the FFTs of the two original signals from the one combined FFT. Since we have six channels of real data coming in with every chunk and need to take the FFT of all six, we use this cycle reducing technique and compute six real FFTs within three complex FFTs.

Here is a diagram showing how the different parts of an input data set change when you take its FFT. Notice how there are 4 distinct parts in the time domain and each pairs with another distinct part in the frequency domain. This allows separation of the 2 real and 2 imaginary input parts in frequency as mentioned in the paragraph above.

DFT Symmetry Figure:

$$x(n) = x^{e}_{R}(n) + x^{o}_{R}(n) + jx^{e}_{I}(n) + jx^{o}_{I}(n)$$



$$X(k) = X^{e}_{R}(k) + X^{o}_{R}(k) + jX^{e}_{I}(k) + jX^{o}_{I}(k)$$

In this figure, $x^{e}_{R}(n)$ refers to the even and real part of the signal x(n). $jx^{o}_{I}(n)$ refers to the odd and imaginary part of x(n).

It should be noted that the DFT's of $x^{e}_{R}(n)$ and $jx^{e}_{I}(n)$ are $X^{e}_{R}(k)$ and $jX^{e}_{I}(k)$, respectively, but the DFT's of the odd parts are reversed, i.e. $x^{o}_{R}(n)$ ---> $jX^{o}_{I}(k)$ and $jx^{o}_{I}(n)$ ---> $X^{o}_{R}(k)$.

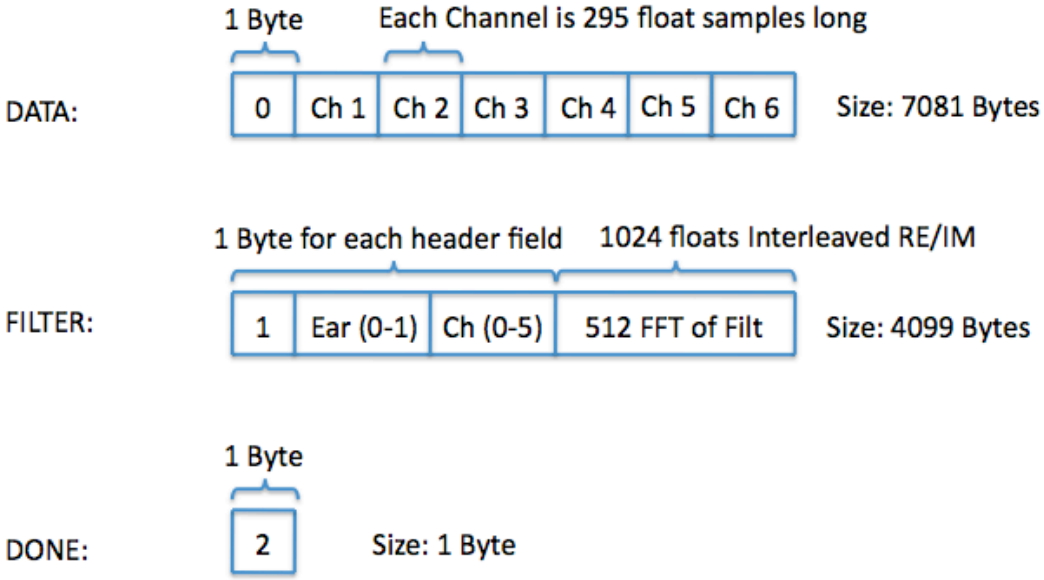
UDP

The last important time saving device we used was UDP. Since this is a realtime streaming project, we have chosen to go with UDP. TCP is slower and is not ideal for streaming applications. The best part about UDP is that if a packet is dropped, it doesn't spend time trying to go back and get the right packet, allowing the current real time packets to keep flowing. There will of course be a glitch in the playback when a UDP packet is dropped, but for streaming purposes this is better than incurring a permanent latency gain with TCP, where it will stop sending by trying to go back in time and fix the problem and then continue from there, and it will never catch back up to real time.

To simplify the tolerance to lost packets, all of our UDP packets are self contained messages (no messages span multiple packets) following one of the following formats:

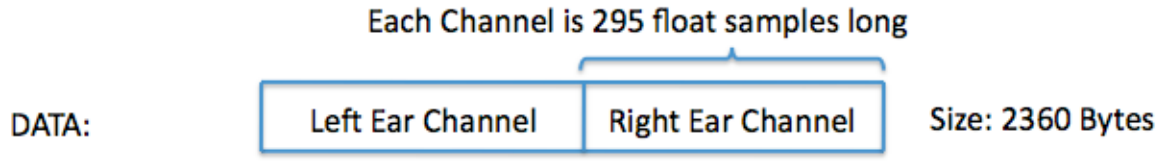
PC Packager to DSK:

Because there are multiple kinds of packets the PC Packager can send to the DSK, we have defined our own header system in the first byte to define the types of packets.



DSK to PC Packager:
These packets aren't checked by the PC Packager. The Packager only waits for ANY packet from the DSK and considers that an "I am ready" message. Because of this, the DSK can send whatever packet it wants. We send junk data 1 byte in length.

DSK to PC Player:
These packets are always data, and so there is no header.



Since we are the first group to work with UDP ever in the history of 18-551, there was a lot of trial and error, and in the end we ran into a lot of problems. See the UDP part of the "Problems" section below for more details.

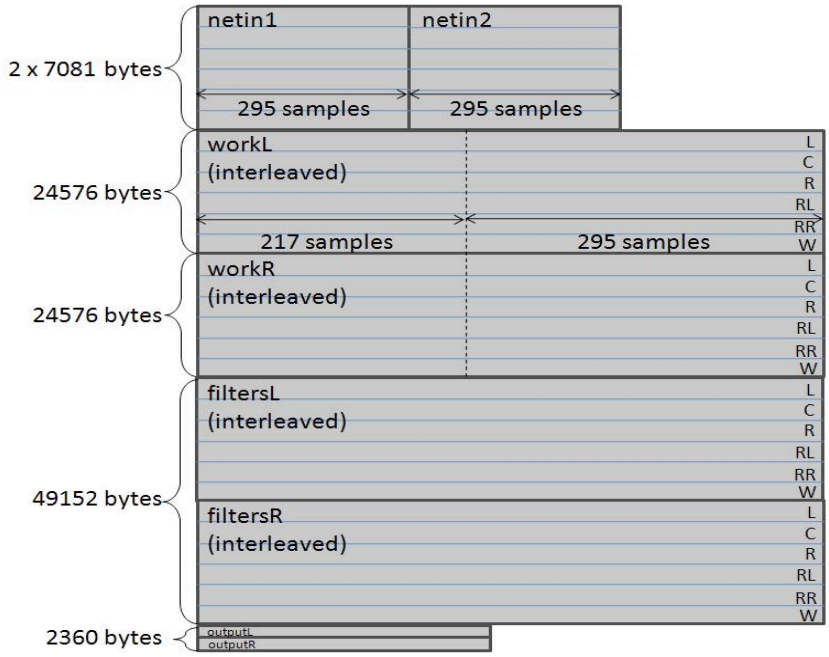
Algorithm Memory and Speed on the DSK

Lots of Calculations...

Before deciding to work with the 48kHz rate, we ran through some numbers and space calculations to make sure everything seemed reasonable and that 48kHz was do-able on the DSK. What follows is a description of our memory usage and speed.

Memory
Starting off, here is a diagram of our internal memory usage and structure:

Total usage: 114826 bytes.
Note that going from 512 to 1024 chunk size would result in not having enough internal memory (once you include the code size and other things)



The netin buffers are used to store the 2 most recent data chunks. We need both because for the overlap save method we need 217 samples of the old data to combine with all 295 samples of the new data. workL and workR are used as workspace areas for the left and right ear. The filtersL and filtersR sections are used to store the 12 HRTF filters that are constantly multiplied by the data. The output buffers are where we write out our processed data to be sent back over the network.

We chose to do everything in internal memory for a couple of reasons. First of all, since we wanted full quality audio, we needed speed. Going to external memory for data is much slower than internal memory accesses, so as long as we could fit our memory usage in the 256k of internal memory, we could get the speed benefits. Of course it is possible to get around this speed problem in external memory by using well-coded EDMA transfers, but this is only helpful when you have a large chunk of data in external memory and you bring in smaller chunks one at a time with EDMA. Since the maximum packet size of UDP is somewhere around 8000 bytes, it doesn't make sense to bring in a huge chunk of data into external memory because we would have to have data span multiple UDP packets, and that requires data flow control that can get quite complex with UDP systems (because UDP packets are not guaranteed to get there). So clearly, purely internal memory with UDP packets was the simple way to go. As a result, there is no paging on the DSK with our project.

Speed and Timing

Before calculating or experimentally determining the number of cycles used by each step of our filtering process, we first calculated how many total cycles were available for use per input chunk if we kept to the restriction of real time playback. As mentioned in the overlap save section, we have 295 output samples for every chunk of input. Therefore, we have 295 samples/chunk * 1 second / 48000 samples = 6.14 ms / chunk. This is how much time is available to do all the processing on one chunk. Next, since the DSK works at 225 MHz, we can determine this limit in DSK cycles by doing 225 MHz x 6.14 ms = 1,382,812 cycles. In other words, if we can do all the network transferring and filtering calculations for one chunk in 1382812 cycles or less, then our project is realizable. Armed with this information, we started experimentally determining how long each of our steps took.

1) Network transfer to DSK

The first step is to transfer 1 PC Packager packet to the DSK and store that in one of the netin buffers. Getting this in cycles is non-trivial with UDP because it is potentially lossy. As a benchmark, we know UDP is faster than TCP, and TCP can go up to 2.5 MB/sec. At that speed, sending the 7081 bytes over TCP at its fastest would take 637290 cycles, which is already a large portion of our 1382812 cycle limit.

To get more specific estimates, we profiled how long it took to send UDP packets on the DSK for different sized packets. (We couldn't get it to profile a receiving function, so sending will have to do). We sent 10000 bytes over multiple UDP packets of size 1000 bytes, 2000 bytes, or 5000 bytes. We found that these yielded transfer rates of 0.12, 2.5, and 6.0 MB/sec. It was clear from this that decreasing the packet size did not result in any gain in transfer speed per packet. This says that the larger the UDP packet the more quickly data can be sent. Since our packets are even bigger than 5000 bytes (7081 bytes), we can expect even better. Compared to the TCP receive rate, this is faster, but this could be because the DSK is better at sending things than recieving things. Remember we found that the DSK is capable of TCP speeds of 10MB/s outgoing vs the 2.5MB/s incoming.

As a bottom line estimate, since we did not successfully profile the receiving side of UDP, we will stick with the TCP estimate for now and assume we can do better with UDP.

2) Interleaving data in preparation for 3 FFTs

This involves writing the 295 samples of the new netin buffer with the 217 most recent samples of the old netin buffer for each of the channels into the workL workspace. This is done so channel1/channel2 are the real/imaginary parts of line 1 of workL, 3/4 are re/im for line 2, and 5/6 are re/im for line 3. We profiled our own code and found this took 7728 cycles.

3) Perform 3 FFTs

Our mixed radix FFT code has different input and output buffers, so the output goes into the bottom 3 lines of workL.

Cycles = 148518 (by code test) - note this is slow compared to assembly (we use C code version)

4) Split into the 6 FFTs of the channels using symmetry arguments (see algorithm details section)

The decomposition of the 6 channels is outputted into workR since workL is full.

Cycles = 15,889 (by code test)

5) Copy the 6 FFT channels of workR into workL in preparation for filter multiplication

Cycles = 15,346 (by code test)

6) Multiply 6 channels of workL by filtersL and 6 channels of workR by filtersR

Cycles = 98,902 (by code test)

7) Add channels of workL and workR in frequency into the top row of each

Cycles = 12,895 (by code test)

8) Perform IDFT on 2 added channels. Result is put in 2nd line of workL and workR.

Cycles = 99012 (by code test) - note this is slow compared to assembly (we use C code version)

9) Copy the real parts of the last 295 samples of the two channels outputed from above into the output buffer, deinterleaving as we go so that the 295 samples of the left channel are followed by the 295 samples of the right channel in memory.

Cycles = 1,710 (by code test)

10) Send data back to PC over UDP

This time we can get more concrete numbers using our profiling test as mentioned in step 1 for sending using UDP. Our sent packets are 2360 bytes long, so using the 2000 byte estimate of 2.5 MB/sec we get 212400 cycles.

Adding up all of these cycle counts, we get an estimated (only estimated because of network transfer parts) 1249690 cycles required per inputed chunk, which is fairly close to our limit, but is still around 100000 cycles below it. Also keep in mind the majority of that is from the transfer to the DSK and thats using the TCP numbers. UDP is supposed to be faster. Additionally, all of these cycle counts were found without doing any loop unrolling or other optimizing methods, which could be another source of improvement in our cycle counts. The bottom line here is that the DSK should be able to keep up with real time and 48kHz, and now it is clear exactly what is going on inside the DSK code-wise.

Database, Signals, and Hardware

Describing the Stuff We Found Somewhere and Used

Database

Our project's only database was for the HRTFs we use for filtering. There were two databases in consideration for our project, MIT Media Lab's KEMAR Dummy-Head Database, and the UC Davis CIPIC (Center for Image Processing and Integrated Computing) Database. The difference in output produced using either database will be minuscule to the human ear, so given the purposes of our project, either one would have been acceptable. But since the UC Davis HRTF Database has better real world data by incorporating many more different types of head shapes and sampling more impulse responses than its MIT counterpart, and because it was based on the MIT database (making it more recent), we decided it was probably better.

To make sure the database met our needs, we tested the database in Matlab with a sample from the matrix as our 6 channel audio source. The results were exactly what we were expecting. In the scene we chose, the bullets flying around Neo seemed much more localized and realistic in space. From there we knew the CIPIC database was exactly what we needed.

It should be mentioned that Group 2 in 2001 used the KEMAR Dummy-Head Database because it was the best and most comprehensive one at the time (The first release of the CIPIC HRTF Database came out 8/12/01).

For the HRTFS in the CIPIC database measurements were taken from a sample of 45 voluntary subjects and includes 250,000 data points for each. In this case, the microphones recorded an HRIR length of 200 for each ear at 44.1 kHz sampling rate and 16-bit resolution. 25 impulse responses were taken in the horizontal direction ranging from -90° to +90° and 50 elevations from -45° to +130.625°. This results in 1,250 positions in spherical space being sampled.

Sample Rates

Because Dolby Digital files have an inherent sampling rate at 48khz, and we value the quality of our output (we don't want to down-sample our data rates unless we have to due to computational reasons), following our plan, all of our calculations will be done in 48 kHz. Multiplying this by the number of channels, our input and output rates are as follows:

Input Data Rate: 288 kHz (1152 Bytes per second)
Output Data Rate: 96 kHz (384 Bytes per second)

Because the Head-Related Impulse Responses from our database are 200 samples long, sampled at 44.1 kHz (0.004535 seconds per sample), we used MATLAB to up-sample them to 48 kHz, yielding 218 samples for each Head-Related Impulse Response (0.004535 seconds per sample).

FFT Code

The assembly and C code we tested and ended up using for our FFT needs was picked up from the TMS320C67x DSP Library Programmer's Reference Guide (Rev. B) (http://focus.ti.com/docs/toolsw/folders/print/sprc121.html). This includes the assembly code for a number of FFT functions, including Radix 2, Radix 4, and mixed Radix, among others. After going through 3 different FFT functions (see the "what didn't work" section) in the end we used the c code version of mixed Radix FFT from this source. The PDF with the all the information on the various algorithms can be found at http://focus.ti.com/lit/ug/spru657b/spru657b.pdf.

Hardware Purchases

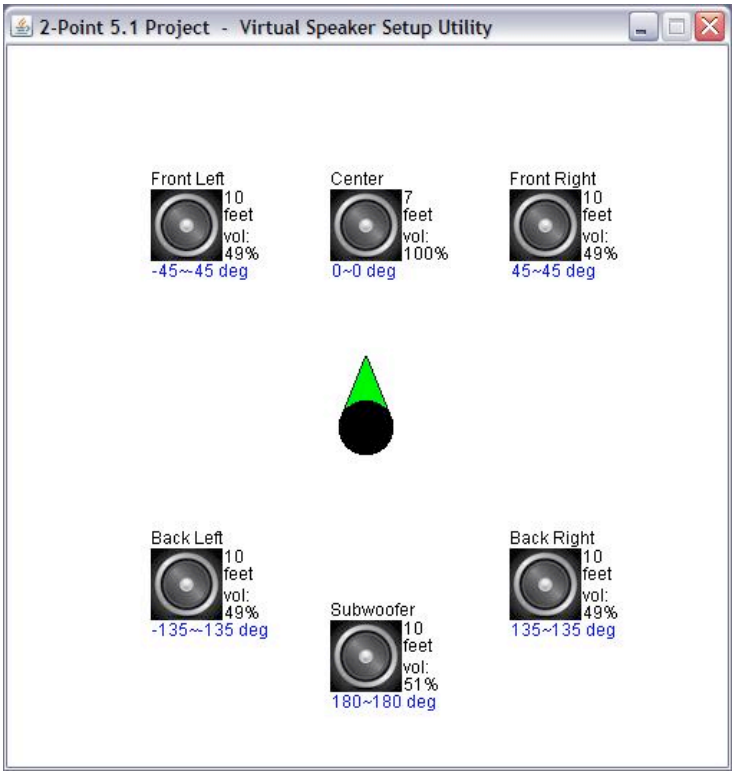
None. Originally we were asking for a nice pair of headphones, but we realized we didn't need them to hear the effects. Therefore, the headphones present in the lab sufficed for our project.

GUI

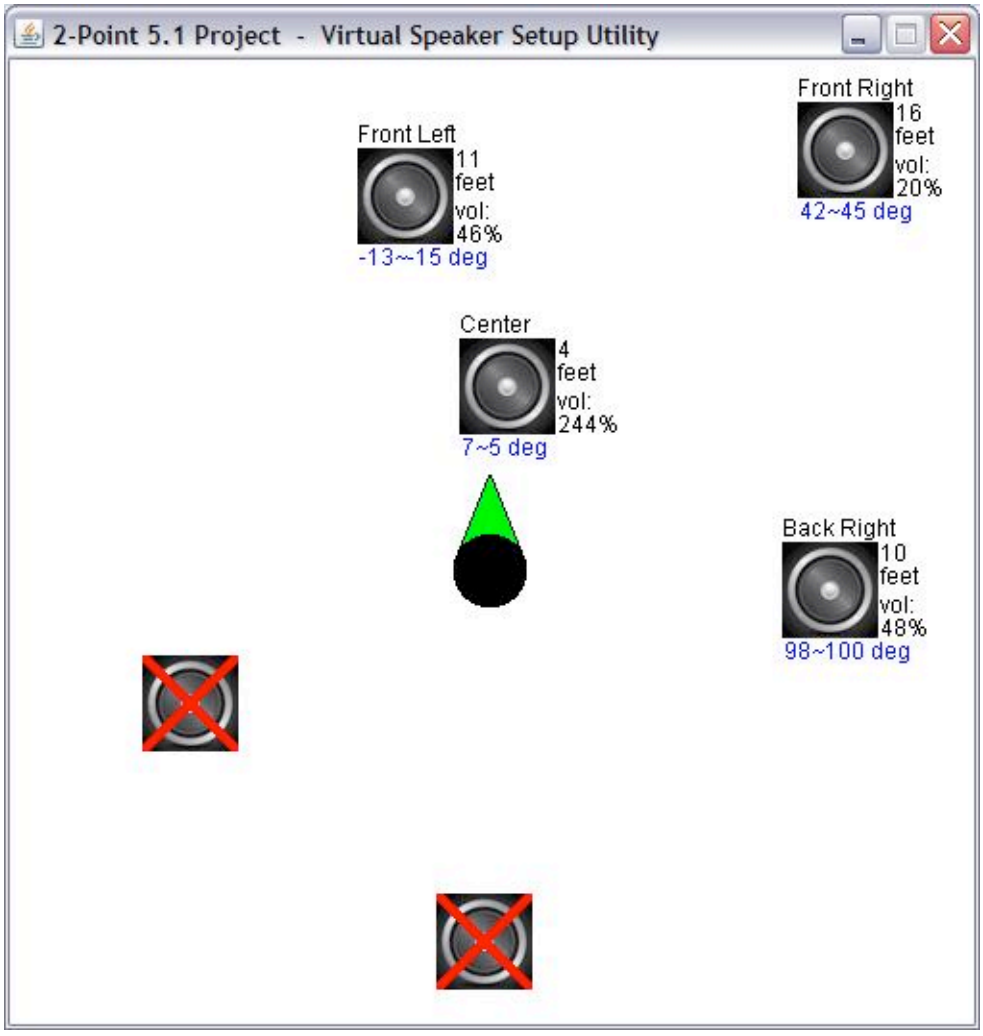
And its Interaction with The PC Packager

Usage

The user starts out being presented by the following default layout. Each of the simulated speakers is represented in space relative to the head (the black and green thing) based on the distances in this diagram:



There are 3 ways to change the environment of what is played back. First of all, you can click and drag a particular speaker so-as to reposition it in space. Secondly, you can click and drag on the empty white space to rotate the entire room (this simulates the listener in the middle turning his head). Lastly, you can right click (without dragging) on one speaker to turn its signal on or off. Using this feature you can listen to as many speaker channels at once as you want. Here is an example of what your setup might look like after playing around with these 3 options:



PC Packager - GUI Interaction

Anytime the GUI is layout is modified, the PC packager becomes aware of this change. It will construct a packet as specified above in the detailed algorithm section where the preFFT'd filter is put into the packet and sent to the DSK the next time the DSK says its ready. The DSK will then hopefully get the packet and update its filters.

The FFT'd Filter coeficients are stored in a pre-created database that we made. To get that database, we simply took our CIPIC database of HRTFs and only took the filters that were at elevation 0 and 180 (level in front of you and level behind you). We then took all of the azimuths at each of these elevations as our data set (after they had been upsampled from 44.1kHz to 48kHz). Next we took the length 512 FFT in matlab of

these filters and stored them so that the java GUI/PC Packager process had access to the data. Then, whenever there is a change in the position of a speaker relative to the head, the program finds the database entry that best matches the angle of the speaker. To simulate distance, the filter is divided in frequency by some constant that is determined by how far away the speaker is. Once this is done, the PC Packager fills the filter update packet and sends it off to the DSK.

What Went Wrong

Stuff We Learned the Hard Way

There were a lot of things that just didn't go well for us. Some of these were as a result of us being stupid and doing something wrong, and some of the problems we encountered were due to non-working code that was not our own that we assumed worked. The following are the biggest things that tripped us up. The problems that were caused by other peoples faulty code (that future groups might want to make note of) are underlined in this section.

Codec

So this isn't really something that went wrong as something we had to change as we went on with the project. Seeing that we were fairly close to our calculated cycle limit for each chunk (back when we did the calculations for assembly code versions of the FFTs) and that was without codec interrupts firing in, which we had heard would bring the DSKs calculations virtually to a standstill, we decided it was better to send our data over the network to the PC and have it play back the audio in real time. Thus, the PC Player process was born.

There were actually a few advantages to letting the PC deal with playback. The first is we could then easily also let it do the recording of the output (for non-realtime playback at a later time). This ended up helping us later when we realized that real time playback was messed up and playing the recorded file let us at least hear what we were doing. Additionally, there is no way we would have been able to play our output through the codec on the DSK and also send the data back to the PC for recording given how close we were to our cycle count limit. Plus in this situation we kept wondering, as long as we were sending the data to the PC for recording, why not also play it there? So in the end playing through the PC player was probably the right way to go.

FFT Algorithms

Before starting the FFT implementation, we didn't think it would be anywhere near as hard as it turned out to be. We ended up spending a lot of time getting our FFTs to work. At first the problems were caused by us.

Back when we thought we were going to use the codec, our first change of plan came when we realized that we needed an interrupt friendly FFT algorithm, and the radix 4 one we used in class was not interruptible, so we decided to go with the mixed radix FFT. Then we found out that the interrupt tolerance of 1 cycle on the mixed radix is still not technically interrupt friendly, so around that time we decided to give up on the codec and go with the PC player. At the same time, we finally came to the realization that our chunk size of 512 was not a power of 4 and we could not use a radix 4 algorithm. We incorrectly assumed that the mixed radix used both and therefore could not work on a size of 512, so we started looking at the radix-2 assembly FFT from the TI website.

Thats when the bugs in the TI assembly code started hitting us. After spending a good amount of time importing code and working out the twiddle factors and such, it turned out the assembly code for the radix-2

FFT wasn't coded correctly. (We determined this with the help of EK, just to be sure). At that point, we gave up on it and decided to move back to the mixed radix (after EK told us that it could in fact do an FFT size of 512). It turned out that TI's assembly code for mixed radix FFT algorithm was also broken (again after EK checked with us). At that point all of our assembly code options were gone, and so we were forced to use the C code version of the mixed radix, which did end up finally working, but it was at least 4 times slower than the assembly code (which we profiled even though it wasn't working properly).

Really Weird Bugs

We had two recurring problems that were probably being caused by the same bug. While the fix was about as simple as possible, finding it took us 2 days of wasted time during the last week when we were crunched for time. The symptoms showed in two ways. In the first, our code would build completely fine and then when we went to run it, it wouldn't even get to the first line of main(). Then when we would comment out some code (we changed which code we commented out with each test, and it was often something as simple as one printf or an if statement) and re-build and test, sometimes it would suddenly work, and others the same thing would happen. The other symptom we saw on the second day was that the network card init function would fail repeatedly, and there was nothing we could do about it no matter how many times we rebooted code composer. It turned out (we think) that the problem was in the cmd file. We had allocated too much space for internal memory (more than it could hold). We changed it to a reasonable value and since then, everything has worked.

UDP

So it turns out there is a fundamental problem/bug in the networking functions in dsknet.h when used with UDP, at least thats what we found. It turns out that when using net_recv() on the DSK to get UDP packets off the network, roughly 19 times out of 20 (at least with our project code) the function would return -1 signifying an error. At first we thought it just always returned -1 (because we never saw the 1 in 20 times it returned something greater than zero) but assumed that it didn't mean anything because the sound coming out of our DSK when we just had it sum the channels (no filtering) was somewhat intelligible. It was only after we looked more closely later at what we were getting that we saw that the packets were actually incorrect most of the time when net_recv() returned -1. We looked at the error code and it said the same thing every time a -1 was returned, namely that there was a CRC check error. CRC errors occur when the transmitted data is incorrect. After studying the contents of the packet more closely, we noticed that every once and a while (roughly 5% of the time we later tested and found out), net_recv() was actually returning the size of a packet (meaning no error).

Confused, and wanting to make sure it wasn't something in our project code, we wrote the simplest UDP sender and receiver code possible between the PC and the DSK. The PC sat in a while loop and endlessly sent a packet that was 15 bytes long where each byte's value was its index from zero to fourteen. The DSK code was sitting in its own loop with a net_recv() function and checking the data of the packets that arrived as well as what the error code was if there was one. It turned out that for every packet that was received from the PC, the net_recv() function returned -1 with an error code of 7 (the CRC check fail one). The really interesting thing was that the data was completely in tact. All 15 bytes were exactly the same as when sent, every single time.

If the data stayed the same as it was sent every time, then technically there would be no problem. We could just ignore the error and use the packet data anyway. But testing the packet data in our project code found that the packets were actually wrong fairly often, and as such, it gave an explanation as to why our output did not sound right. Unfortunately there is no way to fix this problem that we know of. We did not write the net_recv() function, and that seems to be where the problem is. Since there is no way to fix this problem, we just decided to ignore it and take the data out of the packet anyway, but because the data is often wrong, this is probably one of the biggest reasons our output didn't sound right when we demoed.

The other part of UDP that had us messing up was the whole timing of sending and then receiving. Since the calls to send and recv are non blocking, it was easy for our DSK to send an "i'm ready" request to the PC, check the recv function before the PC had sent the packet yet, and then loop back around and send another "i'm ready" message. This caused problems where the DSK would never read the most recent packet sent and instead would jump packets missing upwards of one in every 15 that was sent to it. We eventually fixed this problem by only sending an "i'm ready" packet once after it received a packet, and then only again after a large number of net_recv() loop iterations had occurred with no received packet. Using this technique we were able to show that packets were virtually NEVER dropped (we never saw one drop when we put a breakpoint in the loop on the DSK where it sent ready messages and watched each packet come in one by one after each request) over the network between the PC and the DSK, which is what we expected given what Dan has experienced in 18-345 using UDP.

So in summary with UDP, packets are virtually never dropped on the network so long as you use an "i'm ready" system with the DSK, but the DSK's recv function or some other part of dsknet.h appears to be chronically faulty to the point where the data in the packet doesn't come correctly. In retrospect, given these problems, if we had time it would have been worthwhile to try TCP instead of UDP, and just not worry about the real time results. We would only listen to the recorded file on the PC.

Schedule

I think it was just as clear to others as it was to us that we did not follow our schedule very well. Our original schedule was as follows:

1. Obtain 6 channel wav file database from AC3 audio encoded in DVDs
2. Pick which HRTF filters we will use on each of the 6 channels
3. Decide on a filtering method (We think were going to be using overlap-save)
4. Ensure transferring of data to DSK works with TCP.
5. Ensure audio codec output works.
6. Finalize and optimize filtering algorithm for one channel.
7. Ensure audio codec output works while CPU intensive actions are being performed.
8. Apply filtering technique to ALL channels.
9. Try all of the above with UDP transfer protocol
10. See how well this has been implemented and decide on real-time vs. PC file output
11. Create a GUI to let the user determine the speaker placement

Who Did What

Start Date	End Date	Task Description	Who did it
	10/5	Obtain 48 kHz 6 channel wave file from AC3 audio from DVDs	D
10/13	10/13	Pick HRTF filters	V,Y
10/13	10/13	Decide on filtering type	D,V,Y
10/13	10/19	Ensure transferring of data to DSK works with TCP	D,V
10/20	10/26	Test if the audio codec output works with FFT assembly	Y
10/31	11/1	Decide on a filtering algorithm	D,V,Y
11/10	11/15	Implement filtering algorithm for one channel	D, Y
11/15	11/20	Get 6 channels filtering	D, Y
11/20	11/30	Get it to work with UDP	D, V
11/1	11/30	Work on the GUI	V
12/1	due date	Debugging/Optimization	D, V, Y

In the end, we kept to our different parts, but everything was started the pretty much the week before thanksgiving break (with the exception of the initial testing of algorithms part done by Dan). Here's how it ended up happening:

Planning and organization: All of us
Initial Testing of Algorithms in Matlab and figuring out how to get the data from a wav file: Dan
DSK code and Documentation: Dan and Yiran
PC-side Data Extraction, Networking, Audio Regeneration, Playback, and GUI: Vitaly
Integration and Testing: All of us

As a result of being slightly behind, the filter replacing code on the DSK has not been implemented. We

were going to do that once we got straight playback working, but that never happened, so we never implemented the filter changing stuff DSK-side. The code on the PC Packager however does send the filter packets whenever the GUI is changed. This was not available at our demo but is available now. We might keep working on this and try and get it working during our free time in finals week.

What We Demoed

Stuff That Worked

Somewhat Working Project on DSK

For whatever reason that we can't explain, when we tried to demo the code where the DSK sends ready messages back to the PC Packager, the output from the DSK was nothing more than a bunch of clicks (due to the UDP CRC bug we think), yet when we swapped a version of the PC Packager code that did not wait for the DSK to be read (it just spammed packets over the network to the DSK as it got enough data from VLC), the output of the DSK was somewhat intelligible , but really choppy, as if the DSK could not keep up (probably still had the UDP bug, but less of it for whatever reason).

Our current explanation of the choppiness is to blame the UDP packets that are dropped between the PC Packager and the DSK when there is no ready checking on the PC side. In tests we ran in this situation, the DSK was getting a fairly small percentage of the packets from the PC. The packets were all hitting the DSK at a time when it wasn't ready to receive and as the two parties kept "missing" eachother, the DSK sat idle. In one test we ran, the loop in which the net_recv() function sits on the DSK was looping around over 200 times without having net_recv() pick up a packet, despite the fact that the packets were being sent continuously to the DSK at an extremely fast rate. There was clearly something timing related between the sending and the receiving that didn't work. We think this disconnect between the PC and the DSK caused the output to be choppy when played back.

At the time we did not understand the nature of this dropping of packets between the PC Packager and the DSK. We assumed that the net_recv() function was always picking up packets every single time around the loop. After all, the output was choppy, and the easiest explanation was that the PC was giving the DSK more than enough work, and was sending packets faster than the DSK could read them all to the point where the net_recv() function was always receiving a packet but that still wasn't enough to keep up.

Based on our misjudgment, we created a version of the PC code that downsampled the data before sending it to the DSK, effectively cutting the packet send rate by the downsampling factor, which we varied and tried as different value. What we found was that the output became less choppy as we increased the amount of downsampling, but that no matter how low we went (we went all the way down to 4kHz) the output was still somewhat choppy and distorted. At 4kHz there is no way the DSK still shouldn't be fast enough to send and receive all the data, which led us to eventually figure out the day after the demo about the true nature of the packet loss - that it wasn't because the PC was overworking the DSK, but because the DSK was "missing" the packets sent by the PC as it went around its loop. We have now fixed this problem in our code with the "i'm ready" packets being sent back and the only sending "i'm ready" after many DSK listening loop iterations. Unfortunetely this "i'm ready" version still suffers badly from the UDP CRC bug, which we have not found a way of fixing.

Working Algorithm using Matlab Code

Because our demo did not go as planned and you could not hear the effects of the HRTF filtering through the DSK, we wanted to show what the end result should have sounded like. Thats why we also demoed our Matlab script results where it was easy to discern the effects the HRTF filtering had applied. We played the Dolby Digital Test Loop that played "Front Left, Center, Front Right, Rear Left, Rear Right" and we also played a Clip from The Matrix where Neo is dodging bullets. Additionally, we demoed another

script that used HRTFs in a similar way as our project but only took a mono source and simulated it moving around in space by adjusting the HRTF filter we used in real time. The example played here was a clip of mississippi queen that circled around the subjects head.

Filtering Datapath of DSK

Because there was no way to tell if the filtering was working given the audio state of our output during the demo (which was suffering from UDP issues), we showed our unit test code for the overlap save filtering algorithm. What we do in that code is generate 6 example data sets (we used sinusoids at different frequencies), and put them into the first 3 lines of workL exactly like in step 2 of our DSK memory and speed section steps section above. We then call the functions that do steps 3-9 of our filtering process in internal memory. At the end and along the way we looked at the data in graph form and found it was what it was supposed to be. This should be proof that our filtering process as implemented currently on the DSK is correct.

Future Work

If We Had Time

One thing we would have liked to try is reducing the network transfer time lost by converting our data to floats on the DSK. The WAV files we were using were 16 bits per sample, which means a reduction in size of all network transfers by a factor of 2. This gain would be offset by a loss on the DSK side where it would have to convert everything to floats before it could do anything, but we have a hunch this wouldn't take anywhere near as long as the network savings, and we could save a lot of cycles there. A factor of 2 is huge when our transfers take more than half of our cycle limit per chunk.

If we could squeeze the juice out of the DSK enough, we could model first order room reflections and add a room size and shape dynamic to our GUI. This is kind of a stretch, but its something we could certainly do, especially if we were doing this on the PC.

Again, if we had extra cycles to play with, we could add in a 7th or 8th channel and use some of the more advanced audio formats that actually have that many unique channels.