Virtualized Spatial Audio Simulation and Filtering

for:

Professor Peter Dinda
Computer Science 399
Independent Project

by:

Curtis Barrett
Friday, March 16, 2001

The purpose of this project was to develop software for the simulation of sound pressure waves moving in three-dimensional space and parallelize the code for execution on computing clusters. Unfortunately, only the former was completed. Even so, this project was a good learning experience in a number of ways. Aside from the simulation aspect, I also had the opportunity to experience the joys of building user interfaces in Windows, uncovering various quirks in Visual C++, and discovering the inconsistencies of socket programming across platforms.

The server is unsophisticated. It accepts a single connection at a time, suitable since the simulator runs in serial, consuming massive quantities of CPU cycles. Ultimately, the majority of the server code is just a wrapper for the simulator. It has hooks for configuring the dimensions of the simulation environment and the position of the sound source and listeners[1]. The simulator itself is quite efficient. I minimized the number of flops taking place inside the innermost loop, so the CPU requirements are reasonable for a small environment at a low-to-medium sampling rate. The server code is heavily commented, so further explanations of how it functions can be found there.

For the time being, memory constraints seem to be the larger issue for the simulator. The simulation of a two-meter cubed room at 22050 Hz requires 48 MB of memory. At 44100 Hz, this increases to 380 MB. To simulate my living room (approximately 44 cubic meters) at 44100 Hz would require over two GB of memory. This would definitely be a case for parallelization, and the added CPU power would bring the simulation time down to something somewhat reasonable again.

[1] Support for multiple listeners is not implemented yet, but could be added to the server without too much effort.

Too much of the work done on this project focused on the client. The end result, however, is a clean and usable interface that does most everything it was intended to. The network code and audio filtering code were not trivial, by any means, but were easier to create and debug than the user interface code. The network code is remarkably similar to the server's networking code. The audio filtering code uses Intel's Signal Processing Library to apply FIR filters to audio in real time. It functions admirably for processing monaural sound, able to apply a five thousand sample impulse response to a 22,050 Hz audio stream in real time on a 500 MHz Celeron.

The client also has a graphic display of the generated impulse. Basic as it is now, it could be fleshed out to handle multiple waveforms, display Fourier transforms (the SPL does these), progressively display the impulse as it is simulated (also requires some server work), etc. The impulse can also be exported to a text file, with a sample per row. This can be read into a variety of applications for further inspection.

This software is still in its early stages. Plenty of future work can be done for it. The foremost is parallelization. This would reduce the memory and CPU requirements per machine, but would introduce some communications overhead. For every step of the simulation, each node would need to send the edges of its chunk of simulation space to its neighbors. The time needed to extract this data would also have some impact, and the resulting simulation could actually be slower than serial execution in some cases. If space is divided intelligently (minimize the area of the edges while maximizing the volume of each node's simulation space), the simulation should be able to run better. Dividing space into too many chunks, however, could raise the communications overhead over the simulation time. The simulator, then, needs to be able to fetch information about each

node's memory, CPU and load to tune to simulation for the hardware available. This would not be a trivial task, and could theoretically be handled through the client to a certain extent to give the user some control over the simulation.

Another important step is to support geometry other than an empty cube. This would either require building some sort of basic CAD-like interface or allowing the user to import 3D data from an external 3D application. Once the data is in the simulator, the code would need to perform an extra set of calculations at the end of each step to absorb sound into the geometry of the environment. This could be as basic as setting the pressure to zero inside all the solids in the environment, or could associate some sort of absorption coefficient with solids and allow them to transmit sound. This would allow the simulation of a listener in a different room from the sound source, potentially very useful. This would require some extra work to support in a parallel environment, since the geometry is divided up among the simulation nodes.

Another nice feature would be to generate an arbitrary number of impulses from the simulation. A simulation with two listeners positioned the width of a human head apart would generate impulse that, when used to process sound would give the listener the full feeling of being in the simulated environment. In addition to this, being able to apply these impulses to audio files and generate new audio files from these would be nice. At the very least, the ability to record filtered sound to disk would be a useful feature.

A number of things could still be done to improve the code itself. The client is written in plain Windows GUI code, and is quite a mess in some places. Rewriting it in MFC could help to further modularize it and clean things up. The code is also poorly

commented (especially the user interface code), so it could stand some cleaning and documenting. The server has plenty of checks to keep it running without crashing, but a few bugs are still lingering. The most prominent is handling disconnected clients during a simulation run: this often segfaults in the socket code while trying to send a progress update to a socket that has been destroyed (by the SIGPIPE handler). The simulation should really be moved to a separate thread so it can be stopped more safely, or perhaps decoupled from the client connection entirely. A client could connect, start a simulation, then disconnect and reconnect again later to check the progress or get the impulse.

Overall, this project is a success. Even though I wasn't able to get to parallelization, the resulting client and server work quite well and can certainly act as a launching board for future work. I think I learned more about building user interfaces and networking than I did about virtualized audio, but this breadth of knowledge is certainly the most rewarding aspect of this project.