EFFICIENT FINITE DIFFERENCE-BASED SOUND SYNTHESIS USING GPUS

Marc Sosnick **William Hsu**
San Francisco State University
Department of Computer Science
marc@marcsosnick.com whsu@sfsu.edu

ABSTRACT

Finite Difference (FD) methods can be the basis for physics-based music instrument models that generate realistic audio output. However, such methods are compute-intensive; large simulations cannot run in real time on current CPUs. Many current systems now include powerful Graphics Processing Units (GPUs), which are a good fit for FD methods. We describe an implementation of an FD-based simulation of a two-dimensional membrane that runs efficiently on mid-range GPUs; this will form a framework for constructing a variety of realistic software percussion instruments. For selected problem sizes, real-time sound generation was demonstrated on a mid-range test system, with speedups of up to 2.9 over pure CPU execution.

1 INTRODUCTION

Powerful Graphics Processing Units (GPUs) are now common in the standard graphics cards of most desktop and laptop systems. While earlier GPUs are tailored for graphics processing, recent GPUs from companies such as Nvidia (http://www.nvidia.com) have adopted more flexible architectures to support general purpose computing. Software support for non-graphics computing on GPUs has also improved significantly in the last few years, with environments such as Nvidia's Compute Unified Device Architecture (CUDA) [8] and OpenCL [9]. As a result, there has been much development of general computing on GPUs; many of these projects are documented at http://gpgpu.org.

We have been exploring the use of GPUs for real-time sound synthesis. An obvious question is whether GPU memory bandwidth can efficiently support real-time audio. Another question is whether the GPU architecture can reliably operate under the additional constraints of a real time application. Focus should be on compute-intensive and parallelizable synthesis algorithms, to leverage GPU functionality.

One scenario is to implement many copies of relatively low-cost sound synthesis units on the GPU, mix the outputs down to a few channels, and transfer the mix to the CPU. This is useful for environments such as rendering auditory scenes with multiple sources. We have rather different research goals; our target application involves building a responsive instrument based on a compute-intensive synthesis algorithm.

We have implemented a finite difference-based simulation for a two-dimensional membrane (see [1, 7]), which runs in real time on the GPU; the architecture of the GPU is particularly well suited for this type of algorithm. Finite difference methods are well known as an effective approach for sound synthesis; see for example [2, 7]. Such methods can be a framework for constructing a number of complex software percussion instruments; some simple sound samples can be found at http://userwww.sfsu.edu/~whsu/FDGPU. Finite difference-based sound synthesis for large or fine-grained membranes and plates is too expensive to run in real time on CPUs. Previous studies on audio processing using earlier generation GPUs and software have been mixed (see for example [14, 4]). Our results show that it is now feasible to implement such compute-intensive real-time sound synthesis algorithms on GPUs. In general it should be possible to realize many computationally expensive physics-based synthesis models as real-time instruments on portable systems.

Our paper is organized as follows. Section 2 overviews related work on high-performance audio computing. In Section 3, we describe the finite difference synthesis algorithm we worked with, and our implementation using CUDA. We present experimental results and measurements in Section 4. Conclusions are drawn in Section 5.

2 RELATED WORK

The website http://gpgpu.org is a major clearinghouse for information on general purpose computing on GPUs. Relatively few audio-related projects are documented on the site. [14] implemented seven audio DSP algorithms on a GPU. [11] studied waveguide-based room acoustics simulations using GPUs.

GPUs have been used in the real-time rendering of complex auditory scenes with multiple sources. In [3], the GPU is used primarily for computing particle collisions to drive audio events. [15] uses the GPU for calculating modal synthesis-based audio for large numbers of sounding objects. [13] proposed a method for efficient filter implementation on GPUs, and applied it to synthesis of large numbers of sound sources in virtual environments.

Faust [10] is a framework for parallelizing audio applications and plug-ins; it does not currently support GPU computing.

Our target application is a real-time instrument based on a compute-intensive synthesis algorithm, such as a finite difference membrane model. Bilbao has studied extensively the use of finite differencing for sound synthesis; see for example [2]. Since large models based on finite difference methods are too expensive for real-time performance on CPUs, work has been done for example on FPGA-based implementations [7]. Our approach leverages GPUs that are already common on commodity systems, and does not require custom hardware.

3 FINITE DIFFERENCE ALGORITHM

We simulate a membrane using the finite difference (FD) method of approximation of the wave equation with dissipation in two dimensions as derived by Adib [1]. A square membrane is modeled with a horizontal x-y grid of points. The continuous function $u(x,y,t)$ is defined on the spatial x and y, and time t; u is the vertical displacement at the point (x,y) at time t. The wave equation *with dissipation* is given as:

$$\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}=\frac{\partial^2 u}{\partial t^2}+\eta\frac{\partial u}{\partial t} \tag{1}$$

where η is the viscosity coefficient. Expanding with the truncated second-order Taylor expansion:

$$\begin{aligned}&\frac{u^n_{i+1,j}-2u^n_{i,j}+u^n_{i-1,j}}{\Delta l^2}+\frac{u^n_{i,j+1}-2u^n_{i,j}+u^n_{i,j-1}}{\Delta l^2}\\&=\frac{u^{n+1}_{i,j}-2u^n_{i,j}+u^{n-1}_{i,j}}{\Delta t^2}+\eta\frac{u^{n+1}_{i,j}-u^{n-1}_{i,j}}{2\Delta t}\end{aligned} \tag{2}$$

where, since the grid is symmetric, $\Delta l=\Delta x=\Delta y$, and $x=i\Delta x,\ y=j\Delta y$, and $t=n\Delta t$ [7]. Solving for $u^{n+1}_{i,j}$:

$$u^{n+1}_{i,j}=\left[1+\tfrac{\eta\Delta t}{2}\right]^{-1}\left\{\begin{matrix}\rho\left[u^n_{i+1,j}+u^n_{i-1,j}+u^n_{i,j+1}+u^n_{i,j-1}-4u^n_{i,j}\right]\\+2u^n_{i,j}-\left[1+\tfrac{\eta\Delta t}{2}\right]u^{n-1}_{i,j}\end{matrix}\right\} \tag{3}$$

where, from [5]:

$$\rho=\left(v\cdot\frac{\Delta t}{\Delta x}\right)^2 \tag{4}$$

such that v is velocity of the wave in the medium. For our initial experiments, we treat η and ρ as constants, and used known stable values from Land [5].

In a production system with a variable velocity parameter, it will important to test that the system satisfies the so-called *Courant condition* [1]:

$$|v|\le\frac{\Delta x}{\Delta t} \tag{5}$$

to assure system stability.

We implemented u as three 2-D matrices of single-precision (4-byte) floating point numbers so as to maintain compatibility with Nvidia devices of compute capability 1.2 or lower [8]. We use the leap-frog algorithm to calculate the values at $u^{n+1}_{i,j}$ given the values of $u^{n-1}_{i,j}$ and $u^n_{i,j}$ [1]. Boundary conditions are maintained at each iteration by testing the values of i and j and adjusting $u^n_{i,j}$ appropriately. A scalar gain value is used to either clamp the edge (boundary gain = 0) or allow motion dependent on the adjacent internal grid point times the boundary gain (boundary gain < 1) [5]. Corners are given no special consideration. To obtain different sounds, the values of n (grid size), η, ρ, and boundary gain are manipulated. For example, values of η=2x10^{-4}, ρ=0.5, n = 6, and a boundary gain of 0.75 produces a bell-like tone; values of η=2x10^{-4}, ρ=0.5, n = 16, and a boundary gain of 0 produces a drum like tone. Further examples of this can be found at http://userwww.sfsu.edu/~whsu/FDGPU.

To obtain audio output, the membrane must be excited in some fashion, roughly analogous to striking or plucking the membrane. We use a simple Gaussian impulse to initialize/excite the membrane. $u^{n-1}_{i,j}$ is set to 0, and $u^n_{i,j}$ to a Gaussian impulse, as suggested in [2, 5]. To obtain audio output, a point on the membrane is chosen, and the value for $u^n_{i,j}$ is sampled and scaled at each iteration. For our experiment, the center point of the grid was chosen as the output point.

We coded two implementations of (3), one serial and one parallel. As is typical in real-time synthesis applications, we run the simulation for several time steps and store the generated output samples in the audio output buffer. When the audio output buffer is full, it is handed off to the audio driver for playback. The serial implementation (**Figure 1**), is designed to run on the CPU as in [5, 7]. The outermost loop accumulates output samples in the audio buffer. Then we loop over all the grid points to calculate the elements of the $u^{n+1}_{i,j}$ array. Finally, we update the $u^{n-1}_{i,j}$ and $u^n_{i,j}$ arrays, in preparation for the next time-step. This serial implementation is clearly of $O(n^2)$.

```
For t=0 to t=output buffer size
  For row = 1 to N
    For col = 1 to N
      Update $u^{n+1}_{row,col}$
      If row, col is boundary
        Recalculate boundary point
      If row, col is sample point
        Copy $u^{n+1}_{row,col}$ to output buffer
    End for
  End for
  For row = 1 to N
    For col = 1 to N
      $u^{n-1}_{row,col} = u^{n}_{row,col}$
      $u^{n}_{row,col} = u^{n+1}_{row,col}$
    End for
  End for
End for
End
```

Figure 1. Serial implementation of finite difference membrane simulation.

Our parallel implementation of the finite difference simulation for the GPU (**Figure 2**) is written using Nvidia's Compute Unified Device Architecture (CUDA) extension to C, which allows programmers to take advantage of this architecture. Nvidia's GPU hardware is a SIMT (single instruction multiple threads) architecture using scalable arrays of multithreaded streaming multiprocessors [8]. CUDA divides system hardware into *host* and *device*, where the host is the system (PC desktop or laptop) in which the Nvidia device (or GPU) resides, and the device is the Nvidia GPU on which the parallel program, or *kernel*, executes. The host system first prepares the device and then hands off execution of the kernels to the device. Each kernel is executed on the device in a *thread*, and threads are combined into one, two, or three dimensional *thread blocks*. In a kernel, a thread can obtain its unique *x, y, z* position in the thread block, which is what we use to determine the thread's position when calculating *u*. All threads in a thread block execute simultaneously, but can be synchronized [8].

Memory between the host and device can be independent or integrated with system memory, but in either case are addressed separately on the host and device. On some systems page-locked host memory (called *pinned memory*) can be mapped to the device [8]. Pinned memory simplifies and reduces the overhead of asynchronously transferring results from the device to the host.

In our parallel implementation, each grid point update is mapped to a single thread. A thread determines its position in the grid by finding its 2-D location in the thread block [8]. At each time-step, each thread calculates one update of the $u_{i,j}^{n+1}$ array. As with the serial implementation, each thread checks to see if it is at a boundary; if so, it adjusts the current point. The thread that corresponds to the output point also collects data over multiple time-steps, and updates the output buffer. In order to maintain coherence over time, the threads are synchronized at the points illustrated in **Figure 2**.

```
Calc. row and col from thread index
For t=0 to t=buffer size
  Update $u_{row,col}^{n+1}$
  If row, col is boundary
    Recalculate boundary point
  Synchronize threads
  If row, col is sample point
    Save $u_{row,col}^{n+1}$ in output buffer
  Synchronize threads
  $u_{row,col}^{n-1} = u_{row,col}^{n}$
  $u_{row,col}^{n} = u_{row,col}^{n+1}$
  Synchronize threads
End for
End
```

Figure 2. Parallel implementation of finite difference membrane simulation.

To execute each thread block, the host hands off execution to the device. The simulation runs for several time-steps, and the output buffer is filled with the computation results, after which execution on the device stops. If pinned memory is not supported, the host copies the output buffer to the audio output buffer; otherwise the host passes a pointer to the audio driver using the pinned memory as the audio output buffer. The execution of the thread block is repeated for the duration of the output sound.

We were especially interested in two boundary cases. First, if the output buffer size is small, there will be more calls to execute the grid calculation, creating significant setup overhead. Second, if the grid size is too large, the time that it takes to calculate a grid may push latency past acceptable realtime parameters.

4 EXPERIMENTAL METHOD

4.1 System Configurations

We tested our code on three systems. System 1 was a PC with a 2.5 GHz Intel Core 2 Quad running Ubuntu 9.10 with a 2.6.31-20-generic kernel and an Nvidia GeForce GTX285. System 2 was a Mac Book Air with a 1.86 GHz Intel Core 2 Duo and 2 GB of 1067 MHz DDR 3 RAM running OS 10.5.8 and an integrated Nvidia GeForce 9400M. System 3 was a MacPro with dual 3 GHz Intel Quad-Core Xeon and 5 GB of 667 MHz DDR2 RAM running OS 10.5.8 and an Nvidia GeForce 8800 GT.

These systems represent a good cross-section of available midrange cards. The GTX285 is the most powerful of the three, with 240 CUDA cores running at a Graphics clock of 1.48 GHz. The 8800 GT has 112 CUDA cores running at a Graphics clock of 1.5 GHz. The 9400M is a low-end GPU used mostly in systems with restricted power consumption; it has 16 CUDA cores running at a Graphics clock of 0.80 GHz.

The 9400M and GTX285 both support pinned memory, whereas the 8800GT does not. The 9400M is integrated into the motherboard, whereas the 8800GT and GTX285 both are PCI cards. The 9400M's memory is integrated into system memory, while the 8800GT and GTX285 memory is independent of system memory.

4.2 Software Implementation Details

Our parallel software implementation of the finite difference membrane simulation is written in C++ using Nvidia CUDA (The package is available for download at http://userwww.sfsu.edu/~whsu/FDGPU) We use PortAudio (http://www.portaudio.com) in blocking I/O mode as our cross-platform audio interface.

For both serial and parallel versions, the main loop of the simulation runs for a number of cycles and fills the audio output buffer. Data in the output buffer is then passed on to PortAudio for real-time output or to be stored in a file. On systems with pinned memory, samples generated and stored in the audio output buffer are accessed directly through pinned memory. On systems without pinned memory, data in the output buffer is copied from the device to the host. The PortAudio driver blocks until it has received data [12], thus allowing us to clearly test timing by seeing obvious buffer underrun conditions.

5 EXPERIMENTAL RESULTS

On the three systems we outlined above, we tested the audio output quality for real-time performance, for grid sizes from 15 x 15 to 21 x 21, and audio buffer sizes from 8 to 4096. We discovered that, as expected, the larger the output buffer or the larger the grid size, the better the GPU performed, relative to the CPU on the same system. The predominant problem was jitter [6] caused by buffer underruns. On the GTX285 system, with the parallel implementation on the GPU, we experienced clean output across all grid sizes and audio buffer sizes. However, with the serial CPU code, there was jitter when the grid size was greater than 20 or the buffer size was at 4096 samples or larger. On the 8800GT system, we experienced jitter for both parallel and serial versions, when the buffer size was less than 1024 samples and the grid size at 21 x 21. On the 9400M system, we experienced jitter with both parallel and serial versions, when the buffer size was less than 1024 samples, or the grid size was greater than 17 x 17. On all systems, responsiveness was difficult to evaluate objectively; to fill a buffer of 1024 samples at 44100 Hz, would require approximately 23 ms, which [6] identifies as the threshold for perception of latency. It appears that our parallel finite difference simulation, running on the GTX285 system, can be the basis for a responsive software instrument.

While it is difficult to compare performance on the three systems with different CPUs and GPUs, we set up some simple timing experiments to estimate the efficiency of our parallel implementation. We simulated playing a sample for one second, and repeated this five times. We used the built-in CUDA timers to measure the amount of time it took to calculate the samples and transfer the samples from the device to the system, using pinned memory on systems where that is available, and asynchronous transfers for the system without pinned memory. We made measurements for several audio output buffer sizes, and several grid sizes.

System	Buffer Size (Samples)	GPU Time (ms)	Memory Transfer (ms)	GPU Total (ms)	CPU Time (ms)
GTX285	8	1626	0	1626	3060
	512	1062	0	1062	3032
	4096	1067	0	1067	3102
9400M	8	7251	0	7251	4052
	512	5674	0	5674	4088
	4096	2842	0	2842	4133
8800GT	8	2863	705	3568	2562
	512	2095	12	2106	2518
	4096	2110	2	2112	2539

Table 1. Results for fixed 21 x 21 grid and varying output buffer size.

The results of the tests run on our three test systems, with a fixed grid size of 21 x 21 and varying buffer sizes, are summarized in **Table 1**. *Buffer Size* is the size of the output buffer in samples. *GPU Time* is the total execution time in milliseconds of the kernels on the GPU. *Memory Transfer* is the total time in milliseconds to transfer the output buffer from the device to the host; a memory transfer value of 0 indicates that the device supported pinned memory. *CPU Time* is the total execution time in milliseconds of the serial implementation on the CPU. All timings represent a total time over 5 runs of 1- second output each (i.e. total of 220500 samples).

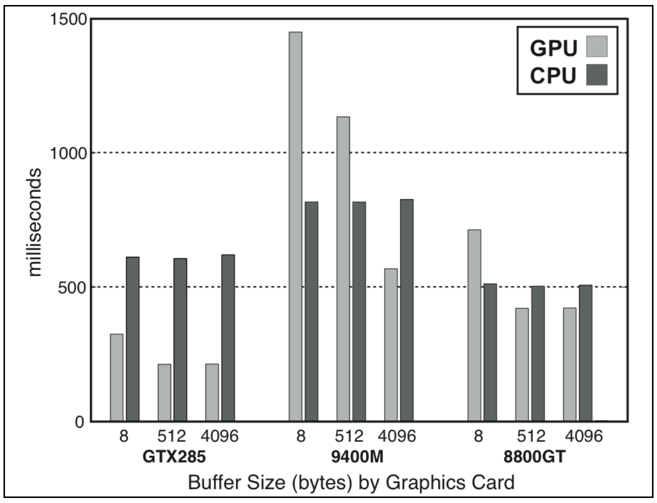

Figure 3. Execution speed with a constant grid size of 21 x 21 points, and varying output buffer sizes.

As can be seen in **Figure 3**, performance on the CPU remains almost constant for all buffer sizes. As the output buffer size increases, generating the same number of output samples requires fewer kernel calls and memory transfers on the GPU; thus the overhead decreases. For the GTX285 system, the performance of the parallel version increased significantly when buffer size increased from 8 to 512, and stayed about constant for larger buffer sizes. The parallel implementation ran faster than the serial implementation, with speedups of 1.2 to 2.9. The 9400M system had the lowest performance of the three. The performance of the parallel implementation increased steadily with larger buffer sizes. For the 8800GT system (no pinned memory), as the buffer size increased, the

System	Grid Size (Points)	GPU Time (ms)	Memory Transfer (ms)	GPU Total (ms)	CPU Time (ms)
GTX285	15 x 15	924	0	924	1577
	18 x 18	984	0	984	2224
	21 x 21	1067	0	1067	3102
9400M	15 x 15	2222	0	2222	1984
	18 x 18	2957	0	2957	3040
	21 x 21	2842	0	2842	4133
8800GT	15 x 15	1411	2	1413	1266
	18 x 18	1743	3	1746	1843
	21 x 21	2110	2	2112	2539

Table 2. Results for a fixed buffer size of 4096 samples, and varying grid size.

overhead for memory transfers decreased as a percentage of total execution time. The parallel code was faster than the serial code only with a buffer size of 512 or greater.

Table 2 summarizes timing estimates with a fixed buffer size of 4096 samples, but with varying grid sizes of 15 x 15, 18 x 18, and 21 x 21. (We were unable to work with larger grid sizes because of GPU memory limitations for our current implementation.)

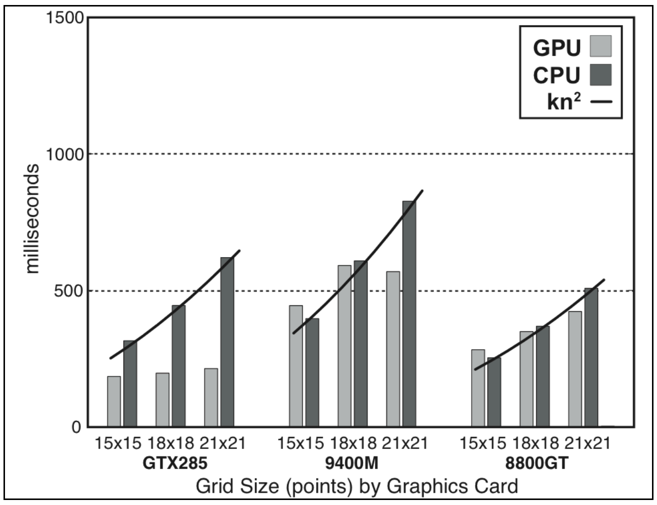

Figure 4. Execution speed with a constant buffer size 4096-samples, and varying grid sizes. For the GTX285, *k*=0.755; for the 9400M *k*=1.0; for the 8800GT *k*=0.629.

As with the previous test, the parallel implementation was faster than the serial on the GTX285 system for all tested grid sizes; it can be seen **Figure 4** that timings for the CPU show an approximate $O(n^2)$ increase with grid size, while GPU timings increase significantly more slowly. With all grid sizes, speedup improved with larger grid sizes. For the 9400M system and 8800GT system, the parallel version was faster for grid sizes 18 and 21, but the serial version was faster for a grid size of 15.

6 CONCLUSIONS AND FUTURE WORK

Our goal for this project was to explore the ability of current mid-range GPU cards to support real-time compute-intensive physics-based synthesis algorithms. We have shown that it is possible to use GPUs to generate real-time audio based on finite difference plate/membrane simulations, but that correct choice of output buffer size and simulation grid size are important. Our straightforward implementation of a parallel finite difference algorithm runs efficiently on our first test system with a GTX285; our less powerful test systems will support adequate performance with selected buffer and simulation grid sizes.

From the results with the 8800GT system, we have shown that memory bandwidth is not a major issue, at least for problems similar to our finite difference code. Newer models of GPU cards that support pinned memory largely avoid the overhead of copying results between the GPU and the host CPU. Larger simulation grid sizes can leverage the parallelism of multiple GPU cores, if the data sizes do not exceed the available GPU memory size. The output buffer size can be increased to reduce kernel call and memory transfer overhead, but at the cost of responsiveness.

Future work will focus on creating a modular production-quality synthesis package using the GPU and finite difference methods, for modeling a variety of percussion instruments. Some limitations of the current implementation must be addressed. Our current version supports only relatively small grid sizes. We are working on distributing the parallel kernel across multiple thread blocks, and using texture memory, to allow for larger or denser grids. Our code is written in the proprietary CUDA extension. We are planning on rewriting the GPU software in the industry-standard OpenCL language [9] and testing it across heterogeneous compute platforms.

7 REFERENCES

[1] A. Adib: "Study Notes on Numerical Solutions of the Wave Equation with the Finite Difference Method," *arXiv:physics/0009068v2 [physics.comp-ph]. 4 October 2000.* Downloaded from http://arxiv.org/abs/physics/0009068v2 on April 15, 2010.

[2] S. Bilbao: "A finite difference scheme for plate synthesis," *Proceedings of the International Computer Music Conference*, pp. 119-122, 2005.

[3] K. van den Doel, D. Knott, D. Pai: "Interactive Simulation of Complex Audio-Visual Scenes," *Presence: Teleoperators and Virtual Environments*, Vol. 13, No. 1, pp. 99-111, 2004.

[4] E. Gallo, N. Tsingos: "Efficient 3D Audio Processing on the GPU," *Proceedings of the ACM Workshop on General Purpose Computing on Graphics Processors*, August 2004.

[5] B. Land: "Finite difference drum/chime," From http://instruct1.cit.cornell.edu/courses/ece576/LABS/f2009/lab4.html, 4/15/2010.

[6] N. P. Lago, F. Kon: "The Quest for Low Latency," *Proceedings of the International Computer Music Conference,* pp. 33-36, 2004.

[7] E. Motuk, R. Woods, S. Bilbao, J. McAllister: "Design Methodology for Real-Time FPGA-Based Sound Synthesis," *IEEE Transactions on Signal Processing*, Vol. 55, No. 12, pp. 5833 – 5845, 2007.

[8] *Nvidia CUDA Programming Guide, version 2.3.1. 8/26/2009.* Downloaded 4/21/2010 from http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/Nvidia_CUDA_Programming_Guide_2.3.pdf.

[9] *Nvidia OpenCL Programming Guide, version 2.3. 8/27/2009*. Downloaded 4/21/2010 from http://www.nvidia.com/content/cudazone/download/OpenCL/Nvidia_OpenCL_ProgrammingGuide.pdf

[10] Y. Orlarey, D. Fober, S. Letz: "Parallelization of Audio Applications with Faust," *Proceedings of the SMC 2009 - 6th Sound and Music Computing Conference*, pp. 23-25, 2009.

[11] N. Rober, U. Kaminski, M. Masuch: "Ray Acoustics using Computer Graphics Technology," *Proceedings of DAFx*, 2007.

[12] B. Roche: Blocking Read/Write Functions. From http://www.portaudio.com/trac/wiki/TutorialDir/BlockingReadWrite, 4/21/2010.

[13] F. Trebien, M. Oliveira: “Realistic real-time sound re-synthesis and processing for interactive virtual worlds,” *The Visual Computer,* Vol. 25, No. 5-7, 2009.

[14] S. Whalen: "Audio and the Graphics Processing Unit," Technical Report, Downloaded 4/21/2010 from http://www.node99.org/papers/gpuaudio.pdf.

[15] Q. Zhang, L. Ye, Z. Pan, "Physically-Based Sound Synthesis on GPUs," *Entertainment Computing - ICEC 2005, Lecture Notes in Computer Science,* Vol. 3711/2005.