Implementing a Finite Difference-Based Real-time Sound Synthesizer using GPUs

Marc Sosnick William Hsu
San Francisco State University, Department of Computer Science
1600 Holloway Ave. TH 906,
San Francisco, CA, 94132, USA
msosnick@sfsu.edu whsu@sfsu.edu

ABSTRACT
In this paper, we describe an implementation of a real-time sound synthesizer using Finite Difference-based simulation of a two-dimensional membrane. 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 demonstrate that it is possible to use this method to create a usable real-time audio synthesizer.

Keywords
Finite Difference, GPU, CUDA, Synthesis

1. INTRODUCTION
Most affordable desktop and laptop systems now include powerful Graphics Processing Units (GPUs). 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. In particular, we are interested in 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.

In previous work, we have shown [12] that it was possible to use the computationally expensive finite difference method to generate sound in real-time. We have subsequently been working to create a usable synthesizer package, *Finite Difference Synthesizer (FDS)*, based on the finite difference method, to generate real-time sound.

NIME'11, 30 May–1 June 2011, Oslo, Norway.

Our implementation uses a finite difference-based simulation for a two-dimensional membrane [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 [3, 7]. Such methods can be a framework for constructing a number of complex software percussion instruments; sound examples generated using the synthesis package will be available 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, 5]). Our earlier results [12] show that it is feasible to implement such compute-intensive real-time sound synthesis algorithms on GPUs. We have since re-designed our software framework to improve the system's use in a real-time performance setting. This paper will focus on software details of our real-time finite difference-based synthesizer for percussion instruments.

Our paper is organized as follows. Section 2 is an overview of related work on high-performance audio computing. In Section 3 we describe the finite difference synthesis algorithm we worked with. In section 4 we discuss details of our software implementation. We present experimental setup in section 5, results and measurements in Section 6. Conclusions are drawn in Section 7. Section 8 outlines possible future directions for the FDS.

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 [4], the GPU is used primarily for computing particle collisions to drive audio events. [16] 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.

Bilbao has studied extensively the use of finite differencing for sound synthesis; see for example [3]. 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. Preliminary results and

measurements were reported in [12]; this paper focuses on details of the current software implementation.

3. FINITE DIFFERENCE AGORITHM

We use the finite difference (FD) method of approximation of the wave equation with dissipation to simulate a membrane 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:

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

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 $u^{n+1}_{i,j}$:

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

where, from [6]:

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

such that v is velocity of the wave in the medium. For our experiments, we treat η and ρ as constants, and used known stable values from Land [6], but allow these to be changed using Open Sound Control (OSC) methods in the synthesis package.

In a final 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 $\eta = 2\text{x}10^{-4}$, $\rho = 0.5$, n = 6, and a boundary gain of 0.75 produces a bell-like tone; values of $\eta = 2\text{x}10^{-4}$, $\rho = 0.5$, 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 [3, 6]. 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 the FDS, the center point of the grid is chosen as the output point.

We used Nvidia's Compute Unified Device Architecture (CUDA) extension to C to implement our parallel implementation of the finite difference simulation for the GPU (Figure 1). 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 of the FD simulation, a single thread is mapped to and calculates each FD grid point. 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 $u^{n+1}_{i,j}$ array. Each thread checks to see if its grid-point is at a boundary; if so, it applies the boundary condition to that point. The thread that corresponds to the output grid-point also updates the output buffer with its vertical displacement over multiple time steps. In order to maintain coherence over time, the threads are synchronized at the points illustrated in Figure 1.

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

Figure 1. Parallel implementation of finite difference membrane simulation.

To execute each kernel, the host hands off execution to the GPU device. The simulation runs for several time-steps, and the output buffer is filled with the computation results, after which execution on the GPU device stops.

4. IMPLEMENTATION

Our software implementation of the finite difference membrane simulation is written in C++ using Nvidia CUDA (The package will be available for download at

http://userwww.sfsu.edu/~whsu/FDGPU). The FDS system uses PortAudio (http://www.portaudio.com) (PA) for real-time audio I/O, liblo (http://liblo.sourceforge.net) for the Open Sound Control (OSC) interface.

The memory of the GPU system is separate from system memory. In order to minimize data transfer latency, it is necessary to hold both the simulation data as well as the buffered audio data in GPU memory. Four grids are kept in GPU memory: FD simulation grids for the current and two past time steps, as well as a Gaussian impulse that is used to excite the membrane. When an excitation command is received, a separate kernel positions and scales the Gaussian impulse, and copies it into the FD simulation grids.

Overall, an FDS-based system acts as an OSC server, waiting for OSC packets to be received, and reacting appropriately to controller input.

4.1 Multithreading

During execution, there are three simultaneous threads running (Figure 2): a primary foreground thread handling control, a Port Audio callback thread [2] for system audio output, and a thread performing the finite difference simulation producing audio data. Communication between the audio data producer (FD Engine) and consumer (PA Callback) is achieved using the PA thread-safe ring buffer.

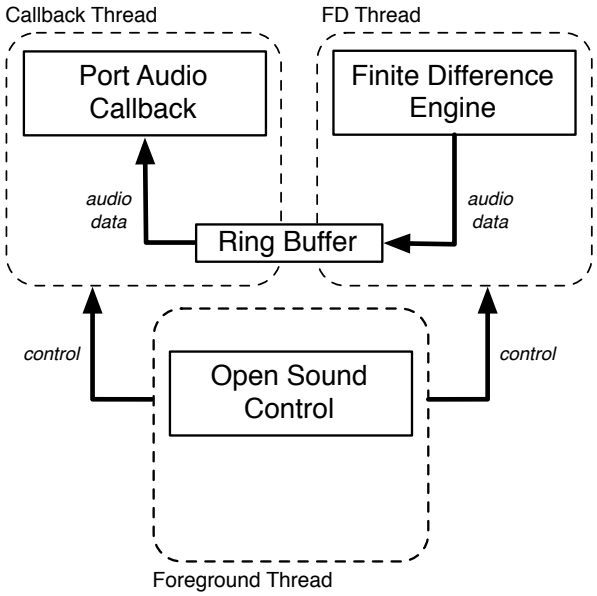

Figure 2. Thread configuration during execution.

4.1.1 Primary foreground thread

In addition to initializing and shutting down the system, the primary foreground thread handles OSC signals and sends user interface commands to the other threads through appropriate semaphores.

4.1.2 Finite Difference Thread

The finite difference simulation is contained in its own thread, and communication with the GPU occurs exclusively in this thread. As mentioned above, control of the simulator such as excitation of the membrane is triggered from the primary thread. After initialization, the finite difference simulation runs continuously, filling the ring buffer with data as space permits. To generate sound, the FD membrane must be excited (perturbed) in some fashion. An arbitrary point on the simulation membrane is used to generate audio output; for the current version of FDS, this is the center of the grid. The value of the vertical displacement of this point at each time step is placed in the audio buffer. The FD kernel (Figure 3) updates the vertical displacement of the grid for a fixed number of timesteps. The displacement of the center point at each timestep is stored into a temporary buffer in GPU memory. The temporary GPU buffer is then copied to the ring buffer in system memory.

Initially all points on the membrane are stationary and have zero vertical displacement. Upon receipt of an excitation command via OSC (e.g. a hit), the primary foreground thread sends a command to the FD thread to excite the membrane. In the FD thread, upon receipt of this command an excitation kernel is called (Figure 3). The excitation kernel copies the Gaussian curve stored in GPU memory to the FD membrane history buffer; this impulse induces vibration in the FD membrane. The excitation kernel can reposition the center of the Gaussian curve to approximate striking (plucking) the membrane at different locations on the surface. The Gaussian curve can also be scaled to simulate harder or softer strikes.

4.1.3 PA Callback Thread

The PA callback thread is a standard audio callback. The callback reads available data in the ring buffer and copies the necessary samples to the Portaudio audio output buffer.

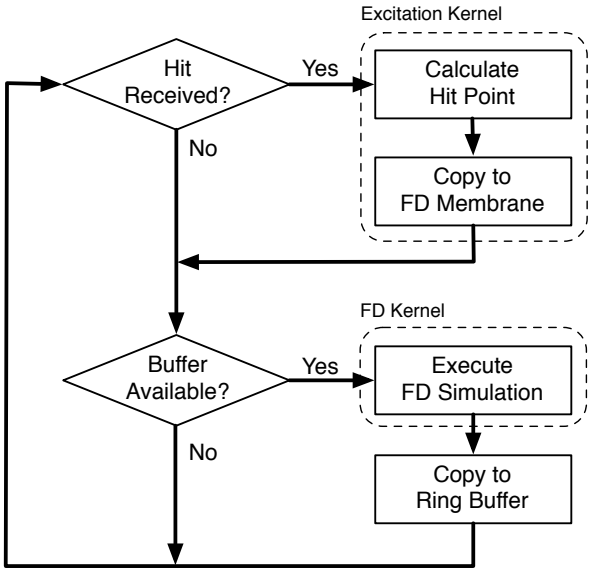

Figure 3. Main FD Thread Loop

4.2 OSC Methods

OSC methods [15] for exciting the membrane using fixed and variable positions, as well as varying amplitude, are available. In addition, FD simulation parameters can be changed using OSC methods, to simulate membranes with different material properties

As discussed in Section 3, for the FD simulation to generate different sounds, the values of n (grid size), η, ρ, and boundary gain are manipulated. For real-time performance, only some of these can be changed in real-time.

For the current implementation of the FDS, after initialization, grid size (n) remains constant. Allocation of both system and GPU memory takes too long to enable reconfiguration in real-time. Once the grid size has been set for a particular sound, it cannot be changed in real-time. The FD simulation parameters η, ρ, and boundary gain (see above) can be changed in real-time; OSC methods are provided for each of these parameters.

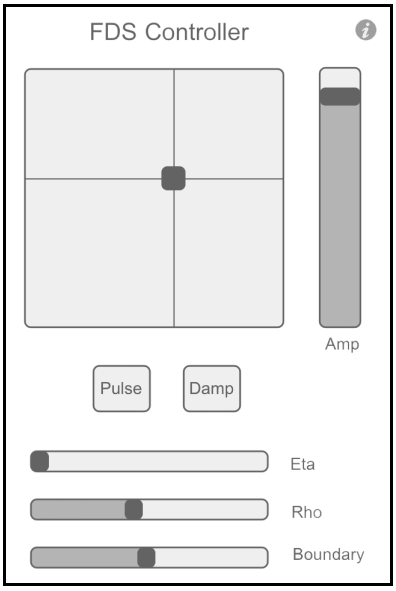

Figure 4. OSC controller interface used in testing.

An OSC controller for the iPhone was developed for use in testing (Figure 4) using TouchOSC (http://hexler.net/). Touching the X-Y pad results in an excitation to the corresponding location on the FD membrane, while the *Amp* slider linearly scales this Gaussian excitation impulse. *Pulse* and *Damp* are momentary pushbuttons; *Pulse* sends a full-amplitude Gaussian impulse to the center of the FD membrane, and *Damp* stops all FD membrane vibration. *Eta*, *Rho* and *Boundary* sliders modulate the parameters described in Section 3.

5. EXPERIMENTAL SETUP

5.1 System Configurations

We tested our system on a MacBook Pro with a 2.66 GHz Intel Core i7, 4 GB 1067 MHz DDR3 RAM, and a GeForce GT 330M GPU running OS 10.6.6.

Timings were taken for two setups. For setup I we held constant a grid size of 21x21 points, and used kernel output buffer sizes of 8, 512, and 4096 entries. For setup II we held the kernel output buffer constant at 4096 entries, and used FD grid sizes of 15x15, 18x18, and 21x21. These values were chosen to correspond to previous tests performed in [12]. In all cases, the ring buffer was guaranteed to have enough space to accept the full contents of the kernel output buffer.

5.2 Testing

For each timing measurement (i.e. each buffer size in setup I and each grid size in setup II), we repeated the following sequence 500 times: run the excitation kernel, check ring buffer space, perform the FD simulation, and copy the FD simulation output to the ring buffer. Timing measurements were averaged over these 500 runs. The built-in CUDA timer routines were used to time memory transfer, excitation, and FD membrane kernel execution times.

A separate test was run with each of the above buffer and grid configurations to ensure that the audio quality was adequate. For this test, the membrane was excited and allowed to play for one second. This was repeated five times. Any audio output buffer underruns were counted; buffer underruns would indicate poor audio quality.

Qualitative testing of the FDS was performed using the OSC controller in Figure 4, changing parameters in real-time.

6. EXPERIMENTAL RESULTS

The results for the timing tests are summarized in Table 1 and Table 2. Total time is the sum of excitation time, finite difference time, and memory transfer time. Buffer sizes of 8, 512, and 4096 samples correspond to audio output of durations 0.181 ms, 11.6 ms and 92.8 ms at a sampling rate of 44,100 Hz.

For the audio quality test, all kernel output buffer and grid configurations produced no audio output buffer underruns.

Satisfactory percussive sounds were produced using the OSC controller interface in qualitative testing. It was found that the FDS's output was sensitive to changes in the FD parameters, especially η and ρ. Recording of some of these tests will be available at http://userwww.sfsu.edu/~whsu/FDGPU.

7. CONCLUSIONS

We have successfully implemented a usable real-time audio synthesizer based on computationally expensive FD simulations. The results of the audio quality tests show that with carefully chosen parameters the FD membrane scheme can generate audio data sufficiently fast to support real-time synthesis. As expected, the majority of the processing time is spent performing the finite difference simulation.

Table 1. Setup I: Results for fixed 21 x 21 grid and varying output buffer size. Timings are averaged over 500 runs.

Buffer Size (samples)	Excitation Time (ms)	Finite Difference Time (ms)	Memory Transfer Time (ms)	Total Time (ms)
8	0.04	0.56	0.02	0.62
512	0.03	6.78	0.01	6.82
4096	0.03	34.31	0.03	34.37

Table 1 shows that as the buffer size increases, the efficiency increases. Time to calculate one sample (time per sample, where 1 sample = 0.026 ms of audio at a sampling rate of 44,100 Hz) for an 8 sample buffer is 0.078 ms, but for a 512 sample buffer it is 0.013 ms, and for a 4096 sample buffer it is 0.008 ms. This decreasing execution time makes sense as the overhead of starting and stopping the simulation and transferring the data is leveraged over a larger buffer size. But this also shows that buffer parameters must be carefully tuned in order to assure adequate real-time performance.

Table 2. Setup II: Results for a fixed buffer size of 4096 samples, and varying grid size. Timings are averaged over 500 runs.

Grid Size (points)	Excitation Time (ms)	Finite Difference Time (ms)	Memory Transfer Time (ms)	Total Time (ms)
15x15	0.03	30.26	0.03	30.32
18x18	0.03	31.81	0.03	31.87
21x21	0.03	34.73	0.03	34.37

Table 2 shows that with an increasing grid size, the simulation efficiency increases. The time to calculate each grid point is 0.13 ms for a 15x15 grid, 0.10 ms for an 18x18 grid, and 0.08 ms for a 21x21 grid.

8. FUTURE WORK

As the majority of execution time is spent in the FD simulation, improvements to this kernel would result in improvements to the overall system.

Other computationally expensive simulations may provide interesting audio results. These simulations would be particularly suited to this synthesis package if the simulation can be efficiently calculated in parallel using GPUs.

To leverage multiple processor environments, current plans include porting the GPU code to the industry-standard OpenCL language [9] and testing it across heterogeneous compute platforms

9. REFERENCES

[1] Adib, A. 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] Bencina, R., and Burk, P. PortAudio – an Open Source Cross Platform Audio API. *Proceedings of the ICMC, 2001*.

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

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

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

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

[7] Motuk, E., Woods, R., Bilbao, S., and McAllister, J. 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] Orlarey, Y., Fober, D., and Letz, S. Parallelization of Audio Applications with Faust. In *Proceedings of the SMC 2009 - 6th Sound and Music Computing Conference*, pp. 23-25, 2009.

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

[12] Sosnick, M., and Hsu, W. Efficient Finite Difference-Based Sound Synthesis Using GPUs. In *Proceedings of SMC Conference 2010, Barcelona*.

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

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

[15] Wright, M. *The Open Sound Control 1.0 Specification Version 1.0*, March 26 2002. From http://opensoundcontrol.org/spec-1_0

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