GPU Based Sound Simulation and Visualization

Torbjorn Loken, Sergiu M. Dascalu, and Frederick C Harris, Jr.

Department of Computer Science and Engineering
University of Nevada
Reno, Nevada, USA
Fred.Harris@cse.unr.edu

Abstract—**As the era of Moore's Law and increasing CPU clock rates nears its stopping point the focus of chip and hardware design has shifted to increasing the number of computation cores present on the chip. This increase can be most clearly seen in the rise of Graphic Processing Units (GPU) where hundreds or thousands of slower cores work in parallel to accomplish tasks. Programming for these chips represents a new set of challenges and concerns. A visualization of sound waves in the room was desired so that phenomena like standing waves could be quickly identified. In order to produce the visualization quickly, some form of acceleration would be required. The GPU was chosen as the accelerator using CUDA to produce the data for the simulation.**

Keywords-CUDA; GPU; Sound; Simulation

I. INTRODUCTION

Computational Acoustics is largely based on the physical modeling of the propagation of sound waves. This propagation is governed by the linearized Euler equations [13]. There have been many methods developed to model the propagation of sound waves; these methods fall into one of two categories: geometric and numeric.

The geometric methods tend to be based on a technique called ray tracing, more commonly used to create computer graphics. These ray tracing methods are able to quickly model the acoustic properties of a room by assuming that high frequency sound waves behave similarly to a ray of light. However, these methods have a critical flaw. Because the wavelengths of audible sounds are on the order of commonly found objects, sound waves exhibit diffraction (bending) where light would not. Because of this recently these geometric methods have fallen out of favor as more accurate models have become accessible.

The numeric methods attempt to directly solve the Wave Equation. The benefit of this approach is that phenomena such as diffraction are accounted for. However, numerically approximating the wave equation can become expensive, especially for FDTD (Finite Difference Time Domain) methods. This is due to the requirements for memory and computation scaling up as the sampling rate desired increases. With the time step T defined as $T = {}^1\!/_{f_s}$ and the grid spacing g defined as $g = (344 * \sqrt{3})/f_s$, it can be seen that as the sampling rate increases not only does the time step of the simulation get smaller, but the size of the grid also increase. These factors combined have meant that until recently FDTD solutions for the acoustic wave equations have been prohibitively expensive.

Recently solutions using CUDA have shown promising results in accelerating 3D FDTD solutions to the wave equation [12, 14].

This paper presents a GPU based approach to sound simulation and visualization in a simple room. The rest of this paper is structured as follows: in Section II we cover the problem background and present information on GPUs, CUDA, and OpenGL. In Section III we present an overview of the application, this is followed in Section IV by a discussion of the implementation. Results are presented in Section V and this is followed in Section VI with Future Work.

II. PROBLEM BACKGROUND

Throughout the history of computing a primary concern for practitioners has been the speed at which results can be computed. Due to this, an active area of research has been the acceleration of computing. The strongest driver for the increase in the performance of computers has historically been the approximate doubling of the number of transistors in any given integrated circuit every two years [10]. This doubling along with Dennard Scaling powered the steady increase in single core processor performance [1]. However, heat and power problems have forced chip manufacturers to develop processors with a larger number of slower cores to use the extra transistors each year [3]. While both Moore's law and Dennard Scaling have improved the computational power of chips, another path to improving the performance of programs has been the addition of specialized hardware to perform computationally expensive tasks. This specialized hardware often takes the form of a coprocessor or a Graphics Processing Unit.

A. Graphic Processing Units

Initially rendering tasks were handled in software using the CPU. Commonly, an FPU was used to help provide newer and better graphical techniques at an acceptable level of performance. However as the focus of the graphical techniques moved from 2D to 3D, specialized coprocessors were designed to keep up with increasingly high standards of performance. The design settled on a massively parallel SIMD architecture in which many pipelines are used to take geometry data, in the form of vertices, and transform it into color values to be displayed on the screen. This SIMD architecture enables many simple processors to work together to produce output simultaneously.

As GPU manufacturers packed more and more vertex and fragment processors into their GPU chips, the appeal of using the GPU for things other than graphics grew. By using the fragment shader in conjunction with framebuffer objects, it was

possible to compute many things in parallel [5,6,9]. This practice called General Purpose GPU (GPGPU) programming allowed many scientific application programmers to accelerate all kinds of calculations. However it required not only knowledge of the problem domain, but also knowledge of the underlying graphic constructs and the API to control it. Despite this limitation, GPGPU enabled some applications to achieve performance gains [4]

B. CUDA

As GPGPU became more widespread GPU, manufacturers began to take note. Starting with the G80 series of graphics cards, NVidia unveiled a new underlying architecture called Compute Unified Device Architecture (CUDA) to ease the struggles of programmers attempting to harness the GPU's computing power [8]. While the new architect did not change the pipeline for graphics programmers, it did unify the processing architecture underlying the whole pipeline. Vertex and fragment processors were replaced with groups of Thread Processors (CUDA Cores) called Streaming Multiprocessors (SM). Initially with the G80 architecture, there were 128 cores grouped into 16 SMs. The Kepler architecture has 2880 cores grouped into 15 SMs as shown in Figure 1. In addition to the new chip architecture, CUDA also included a new programming model which allowed application programmers to harness the data parallelism present on the GPU. The primary abstractions of the programming model are kernels and threads. Each kernel is executed by many threads in parallel; CUDA threads are very lightweight and allow many thousands of threads to be executing on a device at any given time.
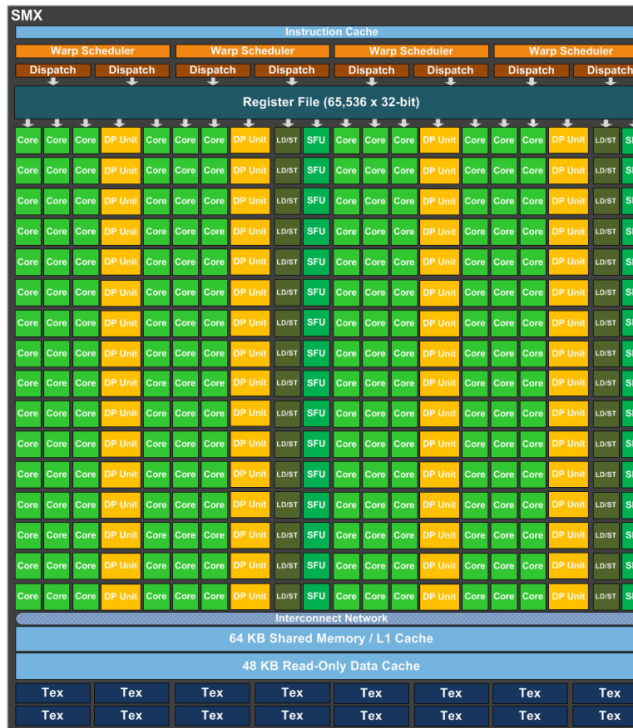

Figure 1.   A block diagram of the Kepler architecture SMX.[11].

C. CUDA Streams

CUDA expose the computational power of the GPU through a C programming model. It additionally provides an API for scheduling multiple streams of execution on the GPU. This allows the hardware scheduler present on CUDA enabled GPUs to more efficiently use all of the compute resources available. When using streams, the scheduler is able to concurrently execute independent tasks. In order to fully use streams for all including memory transfers, the CUDA driver must allocate all memory on the host that will be used for CUDA API calls. This ensures that the memory is pinned (page-locked) so that Direct Memory Access (DMA) can be enabled.

However, using streams is the only way to get the full performance of the GPU using CUDA, and it comes with some concerns. The first concern is that pinned memory cannot be paged out and can therefore impact the amount of paging for any other memory that is virtually allocated. This means that if too much pinned memory is allocated, other components of an application may see a performance loss. In addition care must be taken to order CUDA memory transfers and kernel launches in such a way that the scheduler is able to properly schedule each of the actions concurrently. Additionally some advanced memory features like shared memory or texture memory can become restricted when using streams.

D. OpenGL

OpenGL is an API for rendering 2D and 3D computer graphics. It was originally developed by Silicon Graphics Inc. It provides a portable API for creating graphics which is independent of the underlying hardware. It is widely used for a variety of applications ranging from Computer Aided Design (CAD) to scientific visualizations.

At its core, the OpenGL API controls a state machine. This state machine maintains the hardware and is in charge of ensuring the proper resources are loaded at the proper time. It is important to note that for performance reasons this state machine was not designed to be used with multiple threads. The OpenGL driver can be made to run in a multithreaded way, but the driver does nothing to protect the programmer from race conditions. The API has calls for uploading geometry data, texture data and more to the GPU. In addition, it also exposes a compiler for the GLSL shading language.

The GLSL shading language (shaders) gives the application programmer more control over the functions of OpenGL. The programmer can use these shaders to accomplish advanced rendering techniques such as lighting effects, producing rolling ocean waves or programmatically generating textures or geometry.

One technique of interest to the application discussed in this paper is the rendering of volumetric data. There are many techniques for accomplishing this task [7]. A popular method for volume rendering uses the texture mapping hardware of the GPU and alpha blending to render volume metric data. The technique involves rendering many semi-transparent slices of the volumetric data as shown in Figure 2.
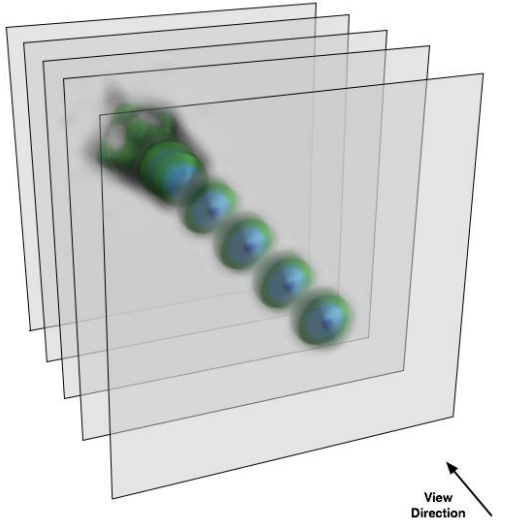
Figure 2. Texture Mapped Volume Rendering: a technique which uses alpha blending to quickly render volumetric data.[2].

III. OVERVIEW

The application presented in this paper uses a 3D FDTD method for simulating sound in a room. A couple of assumptions are made. A simplified model of the sound absorbance is used in lieu of a more computationally expensive one and that all sources of sound are omnidirectional.

The application is structured as a chain of producers and consumers. Once the simulation has begun, the simulation manager begins to produce simulation data. This raw form of the data is both unsuitable for visualization and is also located in a section of GPU memory which the OpenGL driver is unable to access directly. The simulation manager therefore publishes this raw data to a queue for processing. The memory manager takes this raw data, copies it to the CPU and puts it into a form suitable for visualization using OpenGL. These processed frames are published to a queue until they can be uploaded to the OpenGL driver. The renderer is responsible for all the calls to the OpenGL API. After uploading the current frame as a texture, the frame is drawn using texture mapped volume rendering. An overview of this structure can be seen in Figure 3

IV. IMPLEMENTATION

The application is structure into three major components: the renderer, the simulation manager and the memory manager. Each of these components is run on its own thread to allow as many concurrent actions to occur as possible. An overview of each component's execution is shown in Figure 4. Care was taken to design each component in such a way that a component could be redesigned and replaced easily. For instance if the data set would not fit on a single GPU then the simulation manager could be rewritten to accommodate this and the rest of the application could remain unchanged.

A. Render/Main Thread

The first action taken by the main thread is to initialize the state of the OpenGL driver, create a window and initialize a OpenGL context within that window. These actions are all required to begin making any other calls using the OpenGL API. Assuming that there are no errors, the next thing done is the loading of the model information and test sound sample. For this prototype, a test room was made. For the simulation it is important that the model contains both geometry and absorbance information for each point in the grid. The geometry data consists of a 3D array of values encoding whether or that point in the grid is connected its neighbors. Once both the room and sound sample are loaded from disk, the data is used to construct the initial state of the simulation. After the simulation is initialized, both the simulation manager and memory manager are started and the rendering tasks begin.
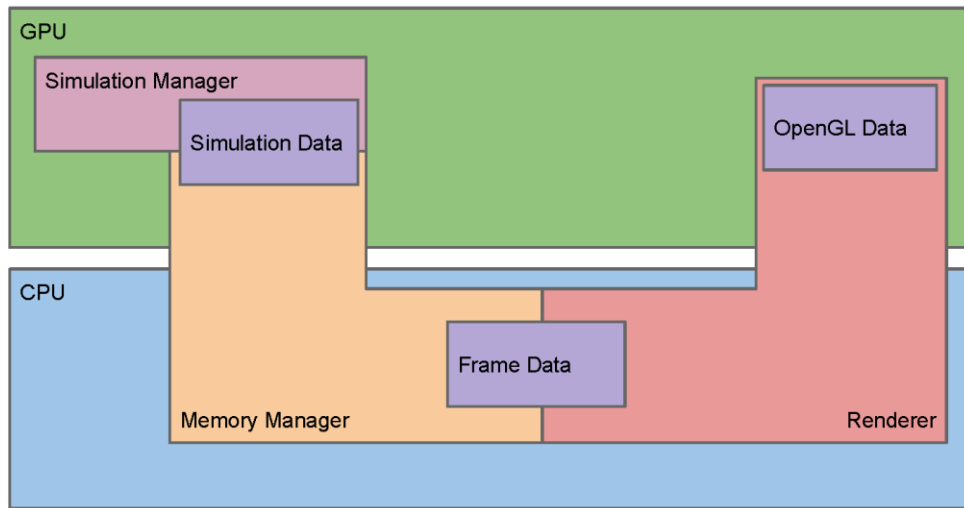

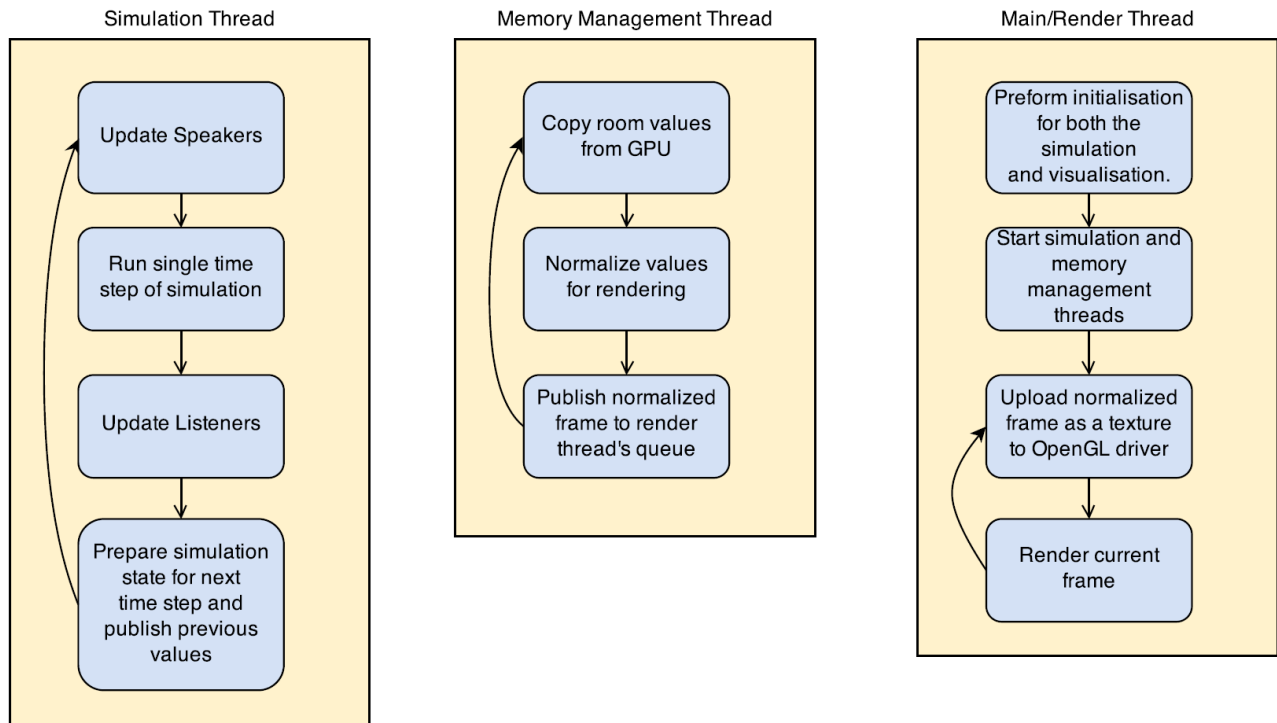Figure 3. An overview of the structure of the application.

Figure 4. An overview of the work done by each of the three threads in the application
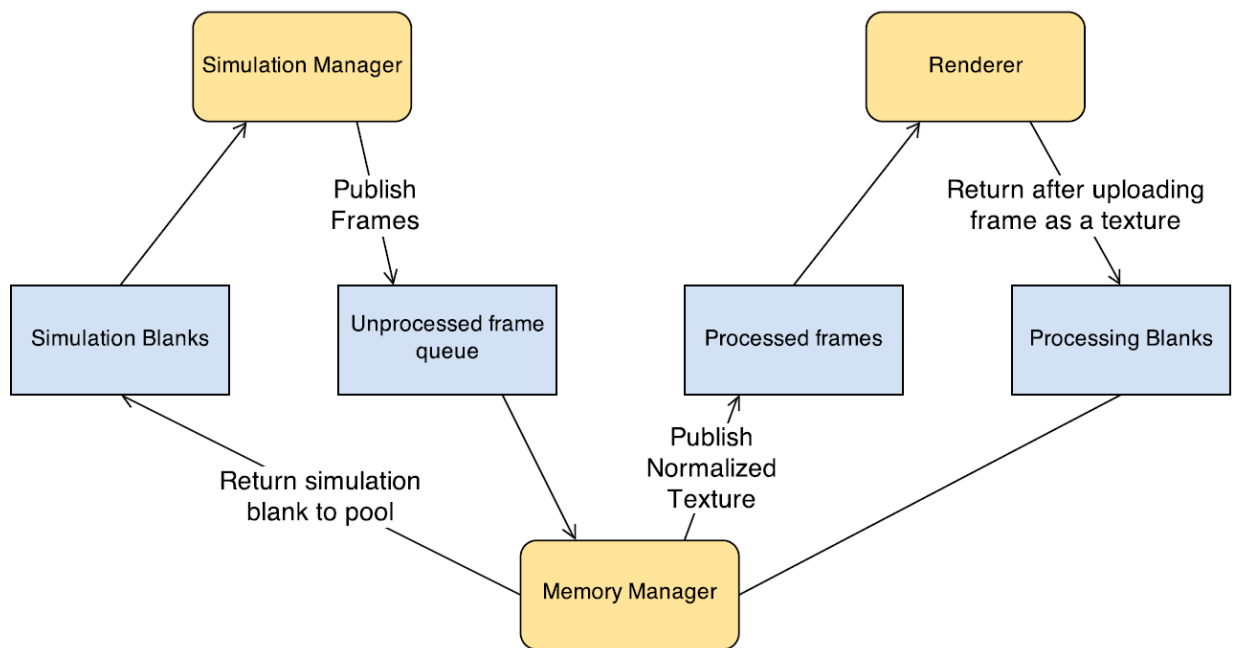

Figure 5. The communication between the three components (shown in yellow) using the various communication queues (shown in blue)

The first of the rendering tasks is to initialize the state of OpenGL, this includes loading the shaders required to perform texture mapped volume rendering as well as preparing OpenGL geometry data required for the technique. Once this initialization has been done, the render thread falls into the render loop.

The render loop primarily consists of two actions: uploading frames to OpenGL memory and performing the volume rendering. Uploading frames to OpenGL memory is the first task done. The renderer checks to see if any processed frames have arrived in its queue, shown in Figure 5 If a processed frame has been published then it is uploaded to OpenGL memory and the now blank frame is published back to the memory manager. The renderer then volume renders the current frame that is in OpenGL. This loop continues until a user inputs the exit command or the simulation reaches its completion.

B. Simulation Manager

The simulation manager's first action is to allocate all the memory required for the simulation (room information, blank simulation state arrays and input/output arrays). It then transfers the encoded grid representing the room, the absorbance values for each point in the grid and the input for the simulation. Once all of the CUDA memory has been allocated and all of the initial simulation state has been transferred onto the GPU, the simulation manager waits to be told to start.

Once the signal to start is received, the simulation acquires three simulation blanks, representing the t_{n-1}t, t_n and t_{n+1} simulation states, from the simulation blanks queue. With that last bit of preparation done, the simulation manager drops into the main simulation loop.

The first step of the simulation is to update all of the input into the simulation. Each input source has a location associated with it that is updated. After that the simulation kernel is run. The kernel uses an equation in the form of the following equation where p_n is the acoustic pressure for t_n to numerically approximate the propagation of the wave.

$$
\begin{aligned}
p_{n+1}(x,y,z) = \frac{1}{3} [& p_n(x+1,y,z) + p_n(x-1,y,z) \\
& + p_n(x,y+1,z) + p_n(x,y-1,z) \\
& + p_n(x,y,z+1) + p_n(x,y,z-1)] \\
& - p_{n-1}(x,y,z)
\end{aligned}
$$

The form of the equation depends on the value of the encoded geometry for that point on the grid. Care is taken to ensure that memory accesses are sequential for the warps assigned to the kernel. The kernel is run in its own stream, which allows the hardware scheduler to schedule both the kernel and any copies that the memory manager is scheduling. If the kernel was not run in its own stream, then the hardware scheduler would not have enough information to schedule the two tasks concurrently. Once the kernel has finished, any listeners present in the model room are updated. The simulation manager then decides whether it's time to publish a raw frame to the unprocessed frame queue as shown in Figure 5.

C. Memory Manager

The memory manager is the simplest of the three components of the application. As seen in Figure 3 the memory manager acts as a bridge between the simulation and the visualization. The memory manage takes the raw frames published by the simulation and normalizes them before publishing them to the renderer. The only major concern for the memory manager is that memory transfers off of the GPU must be run in a separate stream of execution than the simulation kernel. If this is not done, then any memory transfers will block the execution of the simulation kernel.

D. Communication

The application uses three threads to run the three components concurrently. These threads communicate using thread safe queues. The reason that queues were chosen as the data structure for the message passing between the thread is that they both naturally preserve ordering and allow the simulation and visualization to not outpace each other. The application is essentially a producer-consumer problem where both the producer and consumer are fighting for the same resource. If there was not some way to limit which component controls most of the GPU time, then the contention for the GPU would cause problems in either the simulation or the visualization. Additionally, the use of queue as the inter-thread communication medium makes any future attempts to use multiple machines or devices easier by allowing the queue to hide the origin of any information

V. RESULTS.

The prototype was tested on a computer with an Intel Core2 Quad core CPU Q9450 and an NVidia GeForce GTX 480 GPU which contains 15 groupings of 32 compute cores for a total of 480 compute cores and has 1.5 GB of on board memory. Two different test signals were used, one a music sample and the other a 8kHz sin wave. The test room modeled was 12m x 12m x 3m with a pillar in the center of the room. When using the sin wave the simulation modeled standing waves where standing waves would be expected to form. Figure 6 and Figure 7 show the visualization during a test run using the music sample.

Figure 6.   The initial state before simulation initialization

VI. FUTURE WORK

This paper presented an application that benefited greatly from the use of GPU acceleration. The problem presented exhibited data parallelism which assisted in the implementation of the kernel. This acceleration allowed the simulation to run at

a quick enough pace to facilitate the creation of a visualization alongside the simulation.

Despite this, there are issues that should be addressed in future work. Currently, the simulation is limited in size and frequency range due to memory concerns. This could be remedied by replacing the current single GPU simulator with a simulator that uses multiple GPUs if present on the computer or even a clustered simulator. Additionally, the visualization requires that the user has some understanding of the mechanics of wave propagation to be useful. If a user wanted to use the simulator and visualizer to place speakers in a room, it might be more helpful for the visualizer to analyze the frames coming from the simulation to programmatically find good speaker placements.
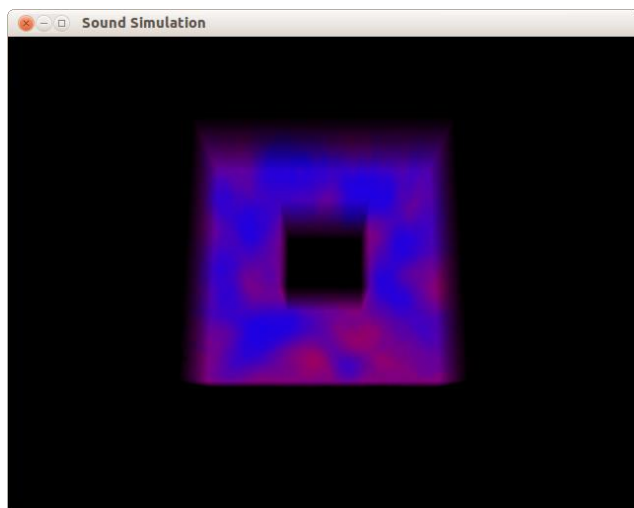

Figure 7.   A rendering from the simulation while running

REFERENCES

[1]   R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. Solid-State Circuits, IEEE Journal of, 9(5):256-268, 1974.

[2]   S. Eilemann. Volume rendering, 2011. http://www.equalizergraphics.com/documents/design/volumeRendering.html.

[3]   H. Esmaeilzadeh, E. Blem, R. St Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In Computer Architecture (ISCA), 2011 38th Annual International Symposium on, pages 365-376. IEEE, 2011.

[4]   K. Fok, T. Wong, and M. Wong. Evolutionary computing on consumer-level graphics hardware. IEEE Intelligent Systems, 22(2):69-78, 2007.

[5]   R.V. Hoang. Wildfire simulation on the GPU. Master's thesis, University of Nevada, Reno, 2008.

[6]   R.V. Hoang, M. R. Sgambati, T. J. Brown, D. S. Coming, and F. C. Harris Jr. Vfire: Immersive wild fire simulation and visualization. Computers & Graphics, 34(6):655-664, 2010.

[7]   M. Ikits, J. Kniss, A. Lefohn, and C. Hansen. Volume rendering techniques. GPU Gems, 1, 2004.

[8]   D. Kirk. NVIDIA CUDA software and GPU parallel computing architecture. In ISMM, volume 7, pages 103-104, 2007.

[9]   D. Luebke, M. Harris, N. Govindaraju, A. Lefohn, M. Houston, J. Owens, M. Segal, M. Papakipos, and I. Buck. GPGPU: general-purpose computation on graphics hardware. In Proceedings of the 2006 ACM/IEEE conference on Supercomputing, page 208. ACM, 2006.

[10] G. E. Moore. Cramming more components onto integrated circuits. Electronics, pp. 114–117, April 19, 1965.

[11] NVIDIA. NVIDIAs next generation CUDA compute architecture: Kepler GK110. http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[12] J. Sheaffer and B. M. Fazenda. FDTD/K-DWM simulation of 3D room acoustics on general purpose graphics hardware using compute unified device architecture (CUDA). Proc. Institute of Acoustics, 32(5), 2010.

[13] C. KW. Tam and J. C Webb. Dispersion-relation-preserving finite difference schemes for computational acoustics. Journal of omputational physics, vol. 107(2):262-281, 1993.

[14] C. J. Webb and S. Bilbao. Computing room acoustics with CUDA-3d FDTD schemes with boundary losses and viscosity. In Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on, pages 317-320. IEEE, 2011.