Editors:
Larry Rosenblum and Simon Julier

Physically Based Sound Synthesis for Large-Scale Virtual Environments

Nikunj Raghuvanshi and Ming C. Lin
University of North Carolina at Chapel Hill

By offering a natural, intuitive interface with the virtual world, auditory display can enhance a user's experience in a multimodal virtual environment and further improve the user's sense of presence. However, compared to graphical display, sound synthesis has not been well investigated because of the extremely high computational cost for simulating realistic sounds. The state of the art for sound production in virtual environments is to use recorded sound clips that events in the virtual environment trigger, similar to how recorded animation sequences in the earlier days generated all the character motion in the virtual world. Although this technique has the clear advantage of being fast and simple, it has two main drawbacks. First, the sound generated is repetitive. Real sounds depend on how objects collide and where impact occurs, and prerecorded sound clips fail to capture such factors.[1,2] Second, recording original sound clips for all the sound events in a virtual environment is a labor-intensive and tedious process.

Physically based sound synthesis, on the other hand, can automatically capture the subtle shift of tone and timbre due to factors such as change in impact location, material property, and object geometry. However, physically based sound synthesis has two computational requirements:

- *An underlying physics engine.* A physics engine informs the sound system of the exact collision geometry and the impact forces involved for every contact in the scene. Many recent commercial physics engines, such as the Havok Engine (see http://www.havok.com), can fill this requirement.
- *Greater computing resources.* Physically based sounds take significantly more computing resources than recorded sounds. Therefore, a brute-force sound simulation will not be able to achieve real-time performance.

In this article, we describe several techniques for accelerating sound simulation, thereby enabling realistic, physically based sound synthesis for large-scale virtual environments.

Overview

Surface vibrations of an elastic object under an external impulse produce sound in nature. These vibrations disturb the surrounding air resulting in a pressure wave that travels outward from the object. If the pressure wave's frequency ranges from 20 to 22,000 Hz, the ear senses it and gives people the subjective perception of sound. The most accurate method for modeling these surface vibrations is to directly apply classical mechanics to the problem, while treating the object as a continuous (as opposed to discrete) entity. This method produces equations that have unknown analytical solutions for arbitrary shapes. One possibility for remedying this problem is to make suitable discrete approximations of the object geometry, making the problem more amenable to mathematical analysis.[3]

Our approach is based on discretizing the object geometry. Given an input mesh comprising vertices and connecting edges, we construct an equivalent spring-mass system by replacing the mesh vertices with mass particles and the edges with damped springs. As we describe in detail, given this spring-mass system, classical mechanics can be applied to yield the spring-mass system's equation of motion

$$\mathrm{M}\frac{d^2\mathbf{r}}{dt^2}+\left(\gamma\mathrm{M}+\eta\mathrm{K}\right)\frac{d\mathbf{r}}{dt}+\mathrm{K}\mathbf{r}=\mathbf{f}$$

where $\boldsymbol{M}$ is the mass matrix, $\boldsymbol{K}$ is the elastic force matrix, and γ and η are the fluid and visco-elastic damping constants for the material, respectively.[2] $\boldsymbol{M}$ is diagonal with entries on the diagonal corresponding to the masses. $\boldsymbol{K}$ incorporates the spring connections between the particles. The variable $\mathbf{r}$ is the displacement vector of the particles with respect to their rest position, and $\mathbf{f}$ is the force vector. Intuitively, the damping constants, spring constants, and masses are determined by the material of the object alone and serve to capture the material's characteristic sound—for example, the thud of a wooden object versus the ringing of a metallic one. The mass and force matrices encode the object's geometry and hence determine the sound's timbre as well as its dependence on the impact forces and positions, which are contained in **f**.

But how exactly does the equation yield an object's sound? Suppose we disturb the spring-mass system by applying an external impulse. This will set the system in motion, and it will start vibrating. The main observation is that the local air-pressure variation in the vicinity of a mass is directly proportional to its velocity. This makes intuitive sense—the faster the object's surface moves, the more it compresses the air near itself, thus raising its pressure. So, we just need to figure out each particle's velocity as a function of time and sum up the pressure contributions from all of the particles. The resulting pressure value as a function of time is the required sound signal.

To obtain the velocity of each particle as a function of time, we must solve the earlier set of differential equations. As we describe in earlier work, we can do this by diagonalizing the force matrix, $\boldsymbol{K}$.[2] The operation's intuitive interpretation is that it translates the original problem in spatial domain to a much simpler problem, in terms of the object's characteristic vibration modes. The sound of each of these modes is a sinusoid with a fixed frequency and damping rate. The key insight is that we can represent all of an object's sounds as a summation of these modes in varying proportions. From a computational viewpoint, the sound system can perform this operation offline as a preprocess, because the mode frequency and damping values depend solely on the object's material properties and geometry. The exact proportion in which the modes are mixed is computed at runtime, depending on the collision impulses and impact position.

A natural question at this stage is how efficient is this naive approach? Typically, the number of modes of an object with a few thousand vertices is in the range of a few thousands, and the procedure we are discussing runs in real time. But as the number of objects increases beyond two or three, performance degrades severely. How can we increase the performance? The key idea is to somehow decrease the number of modes mixed and trick the listener's perception into not noticing the difference.

Exploiting auditory perception

A few techniques use the idea we have discussed to improve performance dramatically and work well in practice for interactive VR applications.

Mode compression

The perceptual study that Sek and Moore describe shows that humans have a limited capacity to discriminate between frequencies that are close to each other.[4] More specifically, if two "close enough" frequencies are played in succession, the average human listener cannot tell whether they were two different frequencies or the same frequency played twice. Table 1 (next page) lists the frequency discrimination at different frequencies. At 2 KHz, the frequency discrimination is more than 1 Hz, which means that a human subject cannot tell apart 1,999 Hz from 2,000 Hz. The frequency discrimination deteriorates a lot as the frequency values increase. Therefore, while synthesizing any sound consisting of many frequencies, we can easily cheat the listener's perception by replacing multiple frequencies close to each other with a single frequency that represents all of them. This streamlining saves computation because mixing one frequency is much cheaper than mixing many—the main idea behind mode compression.

While synthesizing any sound consisting of many frequencies, we can easily cheat the listener's perception by replacing multiple frequencies close to each other with a single frequency that represents all of them. This streamlining saves computation.

After preprocessing an object and extracting its modes, the sound system analyzes the modes to find any that are too close in frequency to each other, as given by the data in Table 1. We sum those that are too close into one mode. The maximum number of modes needed for any object using this technique is always less than 1,000, a number arrived at by purely perceptual considerations and thus completely independent of the object's geometric complexity. In practice, the number of modes after mode compression is typically in the range of a few hundred. This result is a huge gain over the few thousand modes needed for the naive approach. Also, because the sound system computes the modes in a preprocessing step and knows their frequencies in advance, it can perform mode compression as an offline step, saving a lot on runtime cost.

Mode truncation

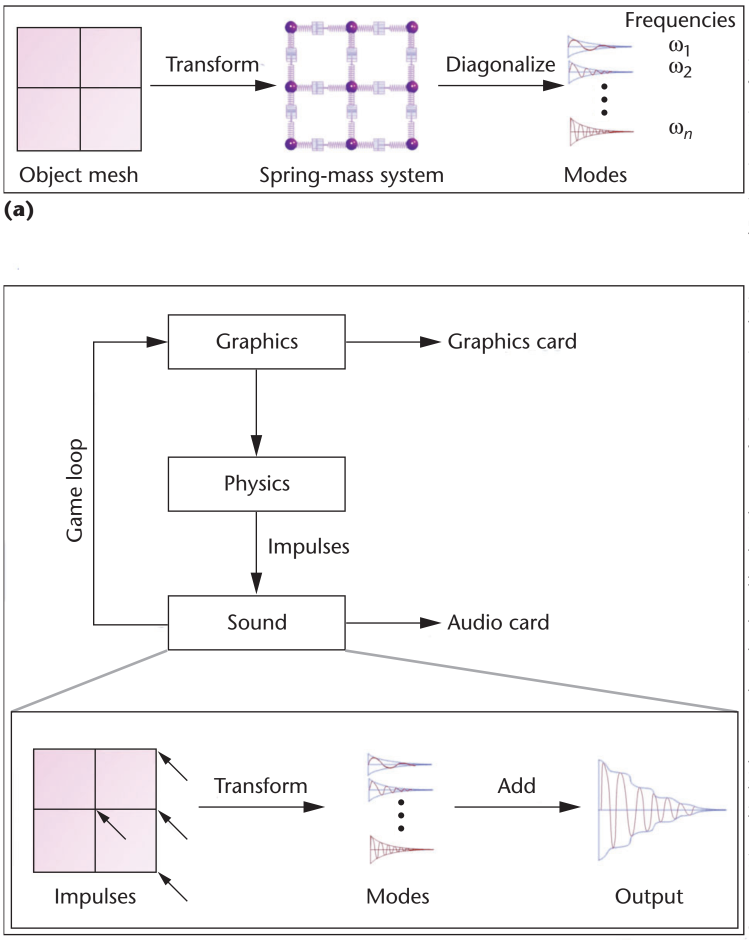
The sound of a typical object on being struck consists of an attack, composed of a blend of high and low frequencies, followed by a sustain, consisting mainly of lower frequencies. The attack is essential to the sound quality, because listeners perceive it as the object's characteristic timbre. For example, imagine striking a bell. The initial sharp sound is the attack, which gives the bell its distinctive timbre, and the subsequent hum is the sustain. Mode truncation aims to stop mixing a mode as soon as its contribution to the total sound falls below a certain preset threshold. This process ensures that the object's attack is captured correctly, which is critical for realism, but it aggressively removes higher frequencies as soon as their contribution reduces to a small value.

Quality scaling

The techniques we have discussed aim to increase the efficiency of sound synthesis for a single object. However, when the number of sounding objects in a scene grows beyond a few tens, increasing individual objects' efficiency is not sufficient.[5] Additionally, it's critical for virtual environments that the sound system has a graceful way of varying quality in response to variable time constraints. We achieve this flexibility by scaling the objects' sound quality. To do this, we control the number of modes being mixed for synthesizing the objects' sound. In most cases of scenes with many sounding objects, the listener's attention is on the objects in the foreground—that is, the objects that contribute the most to the total sound in terms of amplitude. Therefore, if we mix the foreground sounds at a high quality and mix the background sounds at a relatively lower quality, the resulting degradation in perceived aural quality should decrease. Quality scaling achieves this by assigning time quotas to all objects prioritized on their amplitude and then scaling their quality to force them to complete within the assigned time quota. This simple technique performs quite well in practice.

Table 1. Frequency discrimination in humans.

Center frequency (Hz)	Frequency discrimination (Hz)
250	1
500	1.25
1,000	2.5
2,000	4
4,000	20
8,000	88



1 Sound system overview. (a) In preprocessing, the system converts the input mesh for each sounding object to a spring-mass system by replacing the mesh vertices with point masses and the edges with springs. The force matrices are diagonalized to yield the object's characteristic mode frequencies and damping parameters. (b) At runtime, the physics simulator reports the impulses and their locations for each object to the sound system. The system then uses these impulses to find the proportion in which to mix the object's modes, mixes the modes in the determined proportion, and sends them as output to the audio card.

Putting everything together

One of our approach's chief advantages is that it's easy to integrate with any virtual environment that supports physically based interaction. Figure 1 shows a schematic of how our system integrates with a virtual environment. First, every sounding object's mesh passes into the sound system along with its material parameters for preprocessing. The sound system extracts its modes and other information pertinent to sound synthesis. Again, by *mode* we mean a damped sinusoid with a fixed frequency corresponding to a characteristic mode of vibration of the object. Next, the sound system performs mode compression for the object and stores the resulting information for use at runtime. This completes the offline processing.

At runtime, we perform the following steps for each video frame: First, we complete the graphics operations and send the current state of the scene to the graphics card for rendering. Then, we invoke the physics system to time-step the physical simulation. Once the physical simulation is complete, we collect the impact position and impulse values for all objects and pass them on to the sound system. We also inform the sound system of the time it has available for sound synthesis. The sound system does quality scaling for the objects to ensure that the deadline is met and accordingly fixes the number of modes each object must mix for the current video frame. For each object, the sound system samples the object's modes at the required sampling rate (usually 44,100 Hz), scales their contributions depending on the impact and force, and adds up the modes' contributions. The resulting values are the sound samples, which the system sends to the sound card for playback.

Results and demonstrations

We have integrated our sound system with two physics engines and a public-domain graphics engine: Pulsk, which was developed in-house, and Crystal Space (see http://www.crystalspace3d.org), which is one of the most widely used open-source game engines available. Crystal Space can use many of the available open-source physics engines. For our work, we used the Open Dynamics Engine (see http://www.ode.org). All the results were obtained on a 3.4-GHz Pentium 4 laptop with 1 Gbyte of RAM and a GeForce Go 6800 graphics card.

To illustrate the realistic sounds achievable with our approach, we first describe an application that uses Pulsk as the physics engine. We modeled a three-octave xylophone (see Figure 2 on the next page). Each of the wooden keys consists of about 1,000 vertices. The image shows many dice falling onto the xylophone keys to produce the corresponding musical notes. The audio simulation for this scene runs 500-700 frames per second, depending on the frequency of collisions (we define an *audio frame* as enough audio samples to last one video frame). The overall system runs at a steady frame rate of 100 FPS. (To hear an example of sound that our system synthesized, visit http://gamma.cs.unc.edu/symphony.)

To illustrate our system's efficiency, we made a scene with 100 rings falling onto a table in an interval of only a second. We regard this scenario as the worst-case test case for our system, because it's quite rare in a virtual environment for so many collisions to happen in such a short time span. Figure 3 shows the system's performance as a function of time. Because of all the optimization we have discussed, the sound system can stay around 200 audio FPS (see Figure 3, top curve), while a naive implementation would yield only about 30 FPS (bottom curve). Although mode compression and mode truncation do greatly accelerate sound synthesis (middle curve), quality scaling is critical for ensuring that the system maintains high frame rates even when the time constraints are very stringent. That's why, when quality scaling is not used, as in the middle curve, the system's performance dips after 1.5 seconds. This drop doesn't happen when we use quality scaling, as the top curve shows.

To demonstrate our approach's practicality, we integrated our sound system with the Crystal Space game engine. Figure 4 (next page) shows a screenshot from an interactive application using the modified engine. The scene is a typical game-like virtual environment with complex shading involving substantial rendering overhead. The sounding objects in this scene are the red ammunition shells lying on the floor. Each shell consists of a few hundred vertices, and the scene features about 30 sounding shells. The shells make realistic impact and rolling sounds after falling on the floor, and the user can throw in more shells and interact with them in real time. The shells' sound varies depending on how hard the user throws them and how they strike the ground, making the scene much more immersive. This demo runs steadily at over 100 FPS, with the sound system taking approximately 10 percent of the CPU time. The bottom of the figure shows the corresponding sound signal. The peaks correspond to collisions of the shells with the floor. These results clearly demonstrate that today's virtual environments can support physically based sounds for scenes containing numerous sounding objects. (More images are available at http://gamma.cs.unc.edu/symphony.)

Future vision

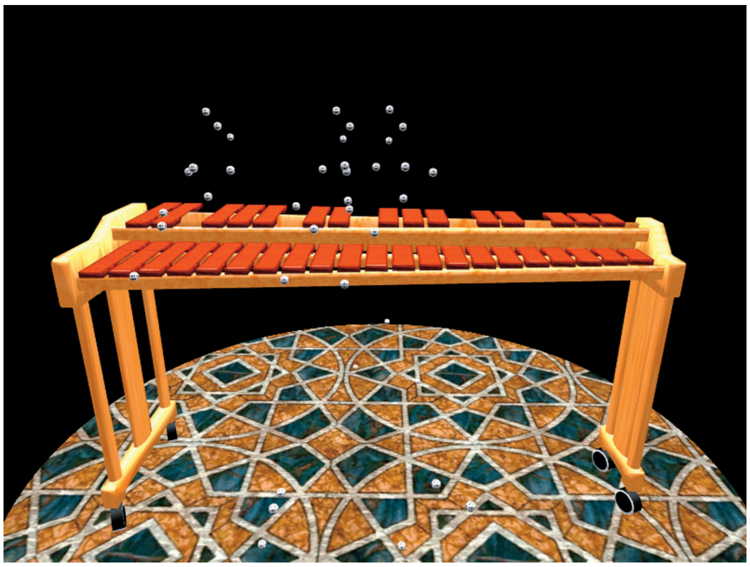
With the methodology and acceleration techniques we have presented, we can simulate sound for a large-scale virtual environment consisting of hundreds of interacting objects in real time, with little loss in perceived audio quality. We expect that developers can apply similar approaches to simulate sliding sounds, explosion noises, breaking sounds, and other more complex audio effects that were difficult to generate physically at interactive rates previously, thus making future virtual environments aurally rich and much more immersive. ■

Acknowledgments

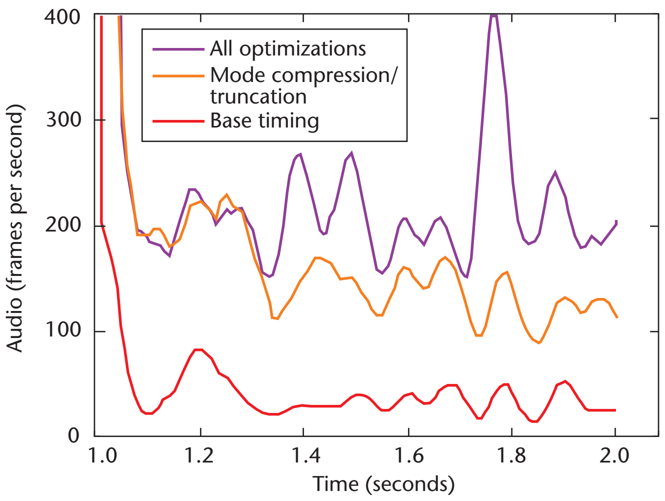
The research project described here was supported in part by the National Science Foundation, Office of Naval Research, US Army Research Office, and Intel.

References

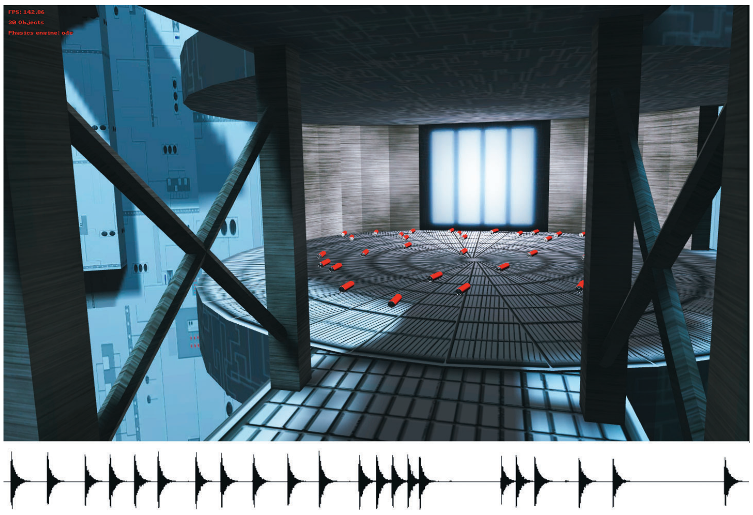
1. K. van den Doel, P. Kry, and D. Pai, "Foleyautomatic: Physically-based Sound Effects for Interactive Simulation and Animation," *Proc. 28th Annual Conf. Computer Graphics and Interactive Techniques* (Siggraph 01), ACM Press, 2001, pp. 537-544.



2 Falling dice on a xylophone. Numerous dice play a song by falling on a three-octave xylophone in close succession. Our system produces the corresponding musical tones at more than 500 frames per second for this complex scene, with audio generation taking 10 percent of the total CPU time.



3 Audio simulation performance. The audio simulation frames per second for a scene with 100 rings falling onto a table in a time span of 1 second, during which almost all the collisions take place.



4 **Real-time sound synthesis in a game-like virtual environment. The red ammunition shells in this screenshot from an interactive VR application demonstrate real-time sound synthesis for numerous objects. The bottom of the figure shows the sound generated for this scene.**

2. N. Raghuvanshi and M. Lin, "Interactive Sound Synthesis for Large Scale Environments," *Proc. ACM Siggraph Symp. Interactive 3D Graphics and Games*, ACM Press, 2006, pp. 101-108.
3. J.F. O'Brien, C. Shen, and C.M. Gatchalian, "Synthesizing Sounds from Rigid-body Simulations," *Proc. ACM Siggraph 2002 Symp. Computer Animation*, ACM Press, 2002, pp. 175-181.
4. A. Sek and B.C. Moore, "Frequency Discrimination as a Function of Frequency, Measured in Several Ways," *J. Acoustic Soc. Am.*, vol. 97, no. 4 , 1995, pp. 2479-2486.
5. H. Fouad, J. Ballas, and J. Hahn, "Perceptually-Based Scheduling Algorithms for Real-time Synthesis of Complex Sonic Environments," *Proc. Int'l Conf. Auditory Display* (ICAD 97), Int'l Community for Auditory Display, 1997, http://www.icad.org/websiteV2.0/Conferences/ICAD97/Fouad.pdf.

Contact authors Ming Lin at lin@cs.unc.edu and Nikunj Raghuvanshi at nikunj@cs.unc.edu.

Contact the department editors at cga-vr@computer.org.