






Real-Time Sound Synthesis and Propagation for Games

Simulating the complete process of sound synthesis and propagation by exploiting aural perception makes the experience of playing games much more realistic and immersive.

By NIKUNJ RAGHUVANSHI, CHRISTIAN LAUTERBACH, ANISH CHANDAK, DINESH MANOCHA, and MING C. LIN

 **T**he believability of a computer game depends on three main components: graphics, behavior (including physics and AI), and sound. Thanks to years of research in computer graphics and the advent of modern graphics hardware, many of today's games render near-photorealistic images at interactive rates. To further increase immersive gameplay, several recent games (such as Valve's *Half Life 2* and Crytek's *Far Cry*) have added an integrated physics and behavior engine to enhance that realism, as objects interact with one another in a more physically plausible way. In contrast, sound generation and propagation have not received as much attention due to the extremely high computational cost for simulating realistic sounds.

The state of the art for sound production in games, even those with integrated physics engines, is to use recorded sound clips that are triggered by events in the

Crowd simulation in an urban scene. (Geometric Algorithms for Modeling, Motion, and Animation Research Group, The University of North Carolina at Chapel Hill.)

game, not unlike how recorded animation sequences were once used to generate all character motion in games. Artist-placed precomputed effects are applied when playing back the sound to mimic spatial effects (such as echoes in a large room) caused by the reflections of sound waves in the scene. Although this technique has the obvious advantage of being simple and fast, it also has major drawbacks. First, the sound it generates is repetitive. Real sounds are dependent on how objects collide and where the impact occurs; prerecorded sound clips fail to capture these factors [7, 9, 10]. Second, recording original sound clips and reverberation filters for all sound events in a game is labor-intensive and tedious. Finally, a static effect that is supposed to represent the acoustic properties of a room can be only a coarse approximation of the real sound and cannot account for many other important contributions that affect immersion in a game environment.

Physically based sound synthesis has clear advantages over recorded clips, as it automatically captures the subtle shift of tone and timbre due to change in impact location, material property, object geometry, and other factors. Physically based sound synthesis has two key requirements:

Physics engine. An underlying physics engine must be able to inform a sound system of the exact collision geometry, as well as the forces involved for every collision, in the scene; many recent commercial games meet this requirement (www.havok.com/); and

Compute power. Physically based sounds take significantly more compute power than recorded sounds. Thus, brute-force sound simulation cannot achieve real-time sound synthesis.

Physically based sound propagation algorithms simulate a sound source in a virtual scene by computing how the sound waves are transmitted in the scene. Broadly, sound propagation simulation involves two different approaches:

Numerical. Numerical methods numerically solve the wave equations to obtain more exact solutions but are impractical for interactive applications due to their demanding computational requirements; and

Geometric. Geometric methods explicitly model the propagation of sound from the source based on rec-tilinear propagation of waves, accurately modeling early specular reflections and transmissions while

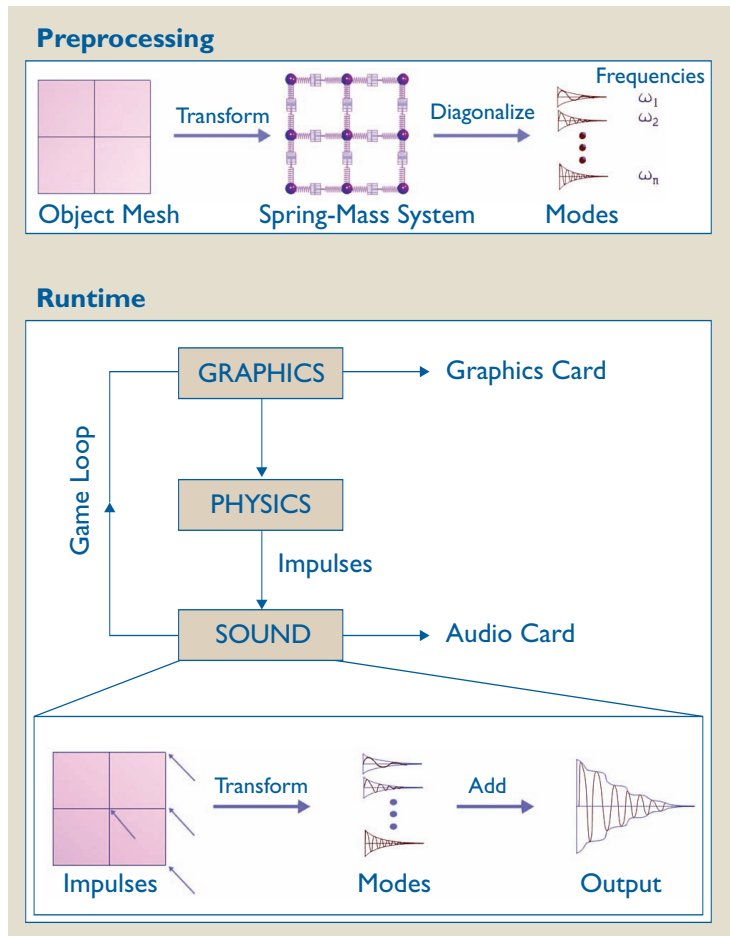


Figure 1. System overview. In the preprocessing step, the input mesh for each sounding object is converted to a spring-mass system by replacing the mesh vertices with point masses and the edges with springs. The force matrix for each spring-mass system is preprocessed to yield its characteristic mode frequencies and damping parameters. At runtime, the game engine's physics simulator reports the impulses and their locations for each object to the sound system. The impulses are then used to determine the proportion in which the object's modes are to be mixed. The mixed modes are then sent as output to the audio card.

accounting for wave effects; solutions for doing this include path-tracing [3] and beam-tracing [2] methods. While the latter are the best of the geometric methods, existing implementations need pre-processed, static acceleration structures for the scene and do not work for either dynamic or general environments (such as those in games).

Here, we describe techniques we've developed to make sound simulation for synthesis, as well as for propagation, much more efficient, thereby enabling realistic sound for games.

SOUND SYNTHESIS

Sound in nature is produced by surface vibrations of an elastic object under an external impulse. The vibrations

disturb the surrounding air, resulting in a pressure wave that travels outward from the object. If the frequency of this pressure wave is within the range 20Hz–22,000Hz, our ears sense it and give us a 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 results in equations for which analytical solutions are not known for arbitrary shapes. To address this problem, one option is to make suitable discrete approximations of the object geometry, making the problem more amenable to mathematical analysis [6].

Our approach (see Figure 1) discretizes an object in the following way: Given an input mesh consisting of 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 described in detail in [7], given this spring-mass system, classical mechanics can be applied to yield the spring-mass system's equation of motion:

$$M \frac{d^2 r}{dt^2} + (\gamma M + \eta K) \frac{dr}{dt} + Kr = f$$

where M is the mass matrix, K is the elastic force matrix, and γ and η are the fluid and visco-elastic damping constants for the material, respectively. The matrix M is diagonal, with entries on the diagonal corresponding to the masses. The elastic force matrix, K , incorporates the spring connections between the particles. The variable r is the displacement vector of the particles with respect to their rest position, and f is the force vector. The damping constants, spring constants, and masses are intuitively determined by the material of the object alone and serve to capture the material's characteristic sound (such as, say, the “thud” of a wooden object, as opposed to the “ring” of a metallic one). The mass and force matrices encode the geometry of the object and hence determine the sound's timbre, as well as its dependence on the impact forces and position of impact contained in the force vector, f .

The equation can be solved analytically by “diago-

nalizing” the force matrix, K , as described in detail in [7]. The intuitive interpretation of the operation is that it translates the original problem in the spatial domain to a much simpler problem in terms of the characteristic vibration modes of the object. The sound of each of

these modes is a sinusoid with a fixed frequency and damping rate. The key insight is that all the sounds of an object can be represented as a mixture of these modes in varying proportions. From a computational point of view, the diagonalization operation can be done offline as a preprocess, as the

frequency and damping of the modes depends solely on the object's material properties and geometry. The exact proportion in which these modes are mixed is computed at runtime and depends on the collision impulses and position of impact. A naive approach to sound simulation would thus consider all the modes of an object and mix them in the appropriate proportions at runtime.

A natural question at this stage concerns the efficiency of the naive approach. Typically, the number of modes of an object with a few thousand vertices is in the range of a few thousand, and the procedure described earlier runs in real time. But as the number of objects increases beyond two or three, performance degrades severely, resulting in pops and clicks at runtime. How can the performance of sound synthesis be improved to achieve real-time performance for interactive gameplay? The key idea is to somehow decrease the number of modes being mixed while tricking the listener's perception from noticing the difference among them.

EXPLOITING AUDITORY PERCEPTION

Two main techniques—mode compression and quality scaling—enhance the efficiency of the approach we've discussed by exploiting human auditory perception:

Mode compression. A perceptual study described in [8] found that humans have a limited capacity to discriminate between frequencies that are close to each other. That is, if two “close enough” frequencies are played in succession, the average human listener is unable to tell whether they were two different frequencies or the same frequency played out twice. The frequency discrimination at different frequencies are listed in Table 1. Note, for instance, that at 2KHz, the frequency discrimination is more than 1Hz. That is, a human subject cannot tell 1,999Hz from 2,000Hz. Observe that the frequency discrimination deteriorates dramatically as the frequencies are increased to higher

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

Table 1. Human frequency discrimination as a function of the center frequency [8]. Our ability to distinguish nearby frequencies deteriorates considerably for higher frequencies, a fact exploited by our approach to improve performance.

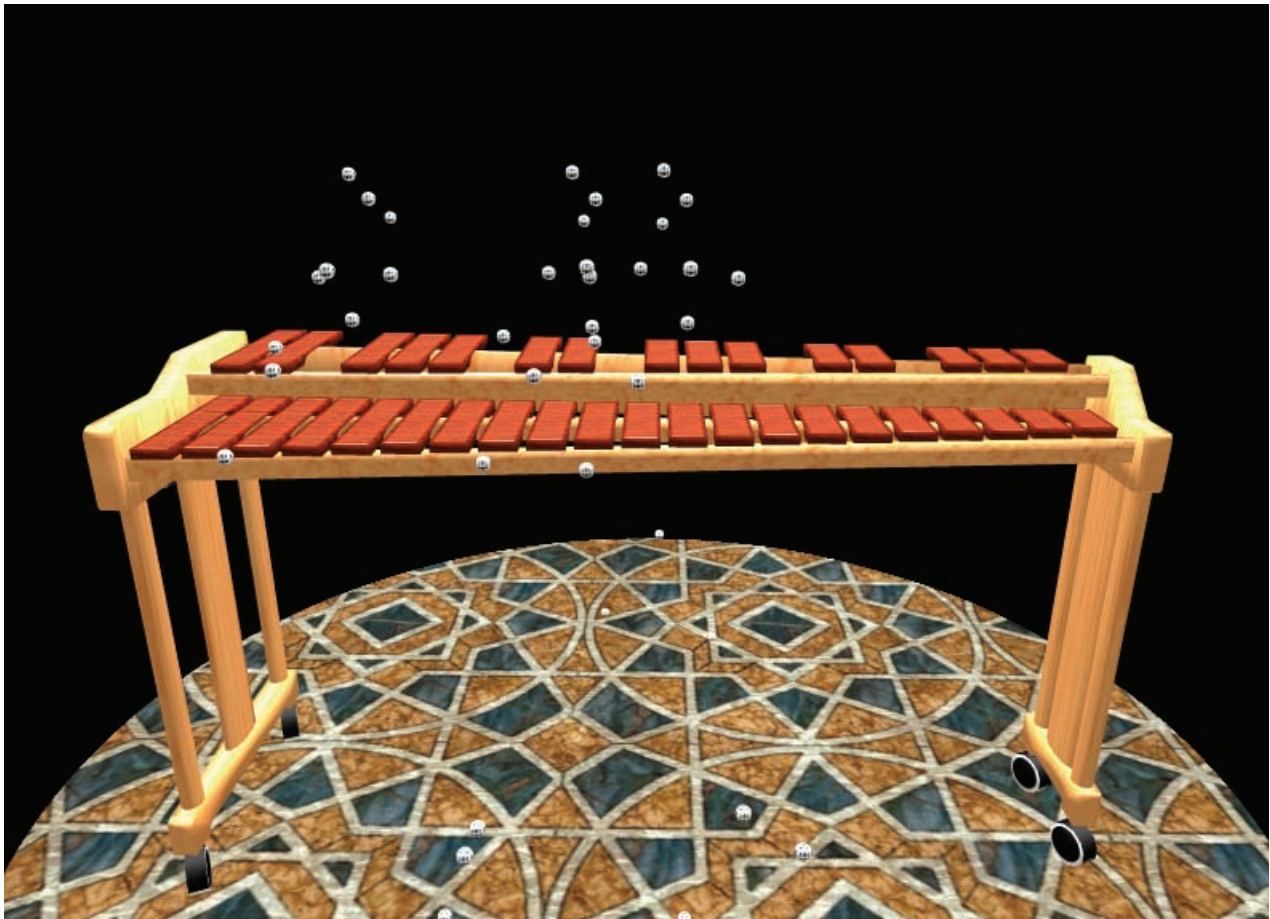


Figure 2a. Falling dice on xylophone. Dice fall on a three-octave xylophone in close succession to play the song “The Entertainer”; see and listen at gamma.cs.unc.edu/symphony. The system produces the corresponding musical tones at more than 500FPS for this complex scene, with audio generation taking 10% of total CPU time.

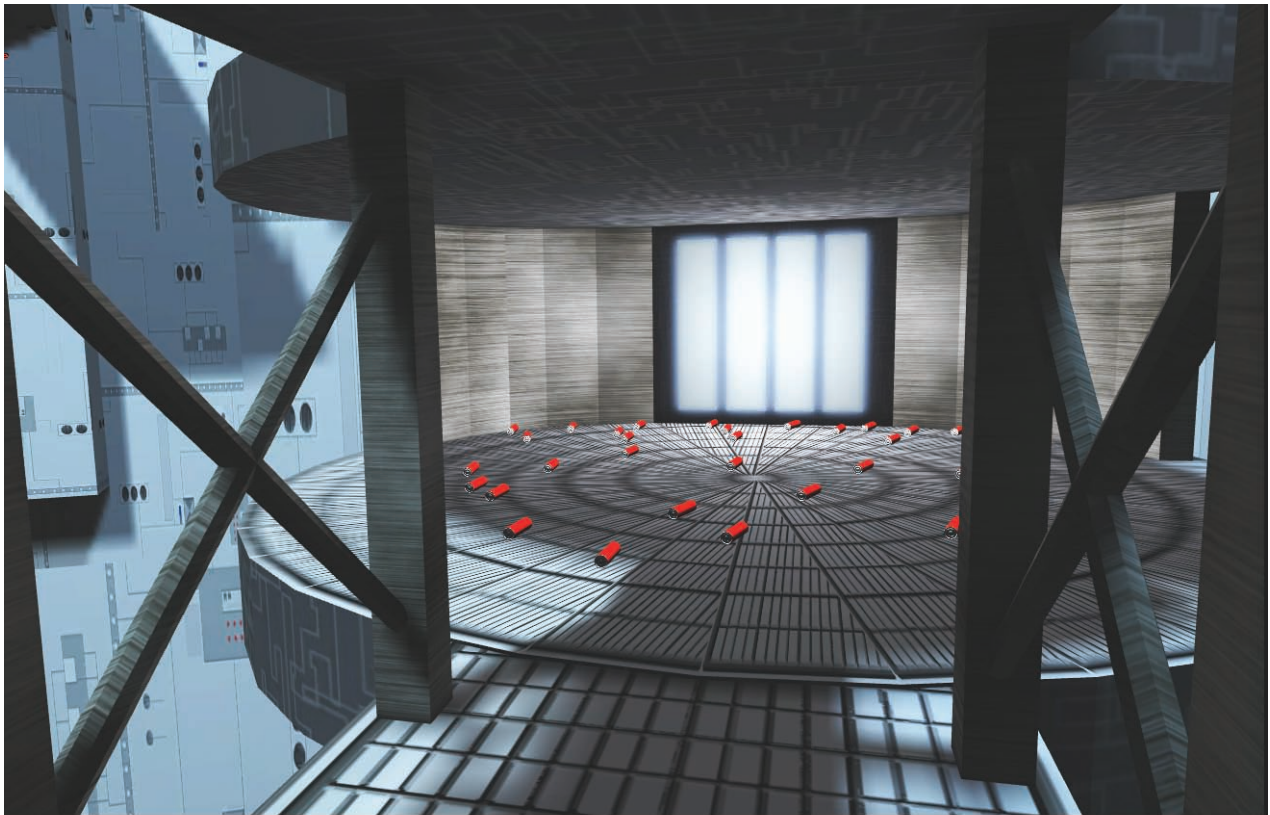
idea behind mode compression and leads to large gains in performance in practice.

Quality scaling. Mode compression aims 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 dozen, increasing the efficiency of individual objects is not sufficient [1]. Moreover, it is critical for a player’s gameplay experience that the sound system employ a graceful way of varying quality in response to variable time constraints. We achieve this flexibility by scaling the sound quality for the objects. This quality is changed by controlling the number of modes being mixed for synthesizing its sound. The main idea is that in most scenes with many sounding objects, the listener’s attention is on

values. 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 one representing all of them. This approach, which saves computation because mixing one frequency is much cheaper than mixing many, is the main

idea behind mode compression and leads to large gains in performance in practice. Therefore, mixing the foreground sounds at high quality while mixing the background sounds at a relatively lower quality should reduce the resulting degradation in perceived aural quality. Quality scaling achieves variable mixing by assigning time quotas to all objects, prioritized on the loudness of the sound they’re producing, then scaling their quality to force them to complete within the assigned time quota—a technique we’ve developed that performs quite well in practice.

We’ve integrated this sound system with two game engines: Pulsk, developed in-house, and the widely used open source Crystal Space (www.crystalspace3d.org/). Crystal Space uses many open-source physics engines; we used the Open Dynamics Engine (www.ode.org/) for our implementation. All results were obtained on a 3.4GHz Pentium 4 laptop with 1GB RAM and a GeForce Go 6800 graphics card. To illustrate the realistic sounds achievable with our approach, we’ll describe an application that uses Pulsk as its game engine. We modeled a three-octave xylophone (see Figure 2a), with each of its wooden keys consisting of about 1,000 vertices. The figure shows many dice falling onto the keys to produce the corre-



sponding musical notes. The audio simulation for this scene runs in the range of 500FPS–700FPS, depending on the frequency of the collisions, where we define an “audio frame” as enough audio samples to last one video frame. The overall system runs at a steady frame rate of 100FPS.

We created a scene with 100 rings falling onto a table in an interval of one second. This scenario can be regarded as the “worst-case” test case for our system, as it is rare in a game for so many collisions to happen in such a short amount of time. Table 2 shows the resulting performance of the system as a function of time. In light of the optimization we’ve discussed here, the sound system is able to stay around 200 audio FPS (top curve), while a naive implementation would yield only about 30FPS (bottom curve).

We integrated our sound system with Crystal Space to demonstrate the practicability of our approach. See Figure 2b for a screenshot from a game application with the modified game engine. The scene is a typical game environment, with complex shading involving substantial graphics overhead. The objects making sounds in the scene are the red ammunition shells on the floor. They make realistic impact and rolling sounds from falling on the floor; the user can toss in more shells and interact with them in real time. This demo runs steadily

at more than 100FPS, with the sound system taking approximately 10% of CPU time.

SOUND PROPAGATION

Our approach for sound propagation is intended as an alternative to beam tracing and is especially well-suited for interactive applications (such as games) while maintaining the advantages of beam tracing. We use bounding volume hierarchy-based ray tracers, since they have been shown to work well for the dynamic and general environments we are interested in [5, 11].

The main difference between our approach and beam tracing is the handling of intersections with the scene. Assume that a beam representing a sound wave hits a triangle of a virtual object. We want a reflection of this beam off the surface, or a secondary beam to represent the reflection, but we are also interested in the remaining part of the beam not hit by the triangle. In beam tracing, these computations are performed using

Figure 2b. Real-time sound synthesis in a game. Screenshot from a game application (using the Crystal Space game engine) demonstrating real-time sound synthesis for numerous objects. The sound being generated is shown below. All red ammunition shells, which were dropped into the scene in real time, are sounding; the user interacts freely with them to produce realistic impact and rolling sounds. The application consistently runs above 100FPS, with sound taking 10% of CPU time; for more images and sounds see gamma.cs.unc.edu/symphony.

exact “clipping” between the beams and the triangles. The intersected part is “cut” out of the beam’s previous shape, with the effect that the beam can have arbitrarily complex, non-convex shapes. This computation can be very slow due to the clipping algorithm and to the difficulty of testing for intersections against these complex beams.

Our frustum tracing approach [4] simplifies this intersection computation by replacing the arbitrarily shaped beams with just one pyramidal frustum that can be uniformly subdivided into sub-frusta. We do not perform exact clipping; instead, we test each sub-frustum against a primitive to see which sub-frusta intersect. Our approach can be interpreted as a discretized version of the clipping algorithm. We represent each sub-frustum by a sampled ray for this purpose and therefore have a group of sample rays representing the whole frustum. This simplification enables game developers to take advantage of recent advances in interactive ray tracing [5, 11], treating a group of rays as a ray frustum to speed operations. Intersection involves testing each ray with the triangle; the test can be inefficient when there are too many sub-frusta. To avoid overhead as much as possible, our method takes advantage of uniform sampling, conservatively determining which parts in frustum space are occluded by the triangle. This approach gives us a quick way to limit the number of rays that need to be tested for intersections.

Since we represent each sub-frustum with a sample ray for this purpose, we introduce an approximation for the intersection test, as well as for the shape of the frustum; the extent of error depends on the number of sub-frusta we generate. Our experiments have shown good convergence rates when increasing the subdivision count for the frustum. Acceptable quality can be achieved with sub-frusta of 4x4 resolution, with only minor quality improvements past 8x8 resolution. In

general, this also has the interesting side effect of producing a general way to trade-off speed and quality by modifying the subdivision factor—important for games where speed is critical.

Using this approach, our algorithm is able to perform the sound simulation many times a second, allowing several orders of reflection in complex game-like scenes that are dynamic and have moving sound sources (see Figure 3). While the overall performance of our approach is still limited by CPU computation power, it is trivially parallelizable and easily integrated into a multi-threaded engine so it runs asynchronously to the other components. Given the trend toward multi-core systems, some part of the overall computational power can then be used for the simulation processes while adjusting the sub-frustum resolution to achieve real-time performance.

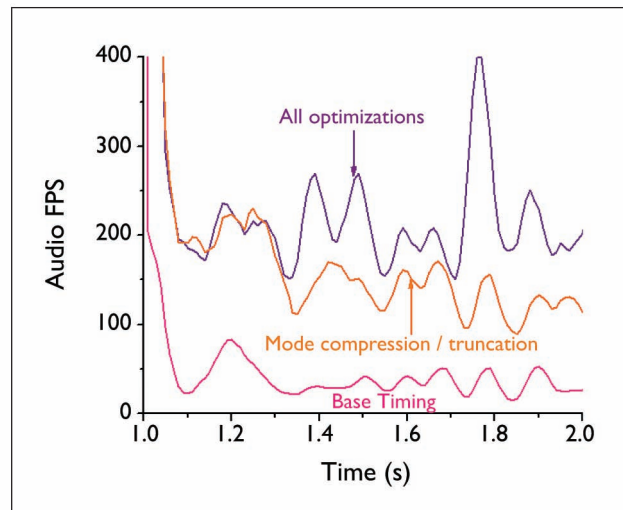


Table 2. Performance. Audio simulation FPS for a scene with 100 rings falling onto a table within one second during which almost all the collisions take place. The bottom-most plot is the FPS for an implementation using none of our acceleration techniques. The topmost plot is the FPS with mode compression, mode truncation, and quality scaling. FPS stays near 200, even when the other two curves dip due to numerous collisions during 1.5–2.0 seconds.

CONCLUSION

This methodology, combined with the acceleration techniques we’ve described, make it possible to simulate sound for large-scale game environments containing thousands of triangles and hundreds of interacting objects in real time with little loss in perceived audio quality. We expect that similar approaches can be applied to simulate sliding sounds, explosion noises, breaking sounds, and other more complex audio effects otherwise difficult to generate physically at interactive rates. Our sound synthesis techniques, in combination with interactive sound propagation, make it possible to fully simulate a sound, from its creation to how it is perceived by the listener, making future games aurally rich and, as a result, much more immersive.

REFERENCES

1. Fouad, H., Ballas, J., and Hahn, J. Perceptually based scheduling algorithms for real-time synthesis of complex sonic environments. In *Proceedings of the International Conference on Auditory Display* (Palo Alto, CA, Nov. 2–5). ICAD, 1997, 1–5.
2. Funkhouser, T., Carlbom, I., Elko, G., Pingali, G., Sondhi, M., and West, J. A beam-tracing approach to acoustic modeling for interactive virtual environments. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)* (Orlando, FL, July 19–24). ACM Press, New York, 1998, 21–32; doi.acm.org/10.1145/280814.280818.
3. Krokstad, A., Strom, S., and Sorsdal, S. Calculating the acoustical room response by the use of a ray tracing technique. *Journal of Sound and Vibration* 8, 1 (July 1968), 118–125.
4. Lauterbach, C., Chandak, A., and Manocha, D. Interactive sound rendering in complex and dynamic scenes using frustum tracing;

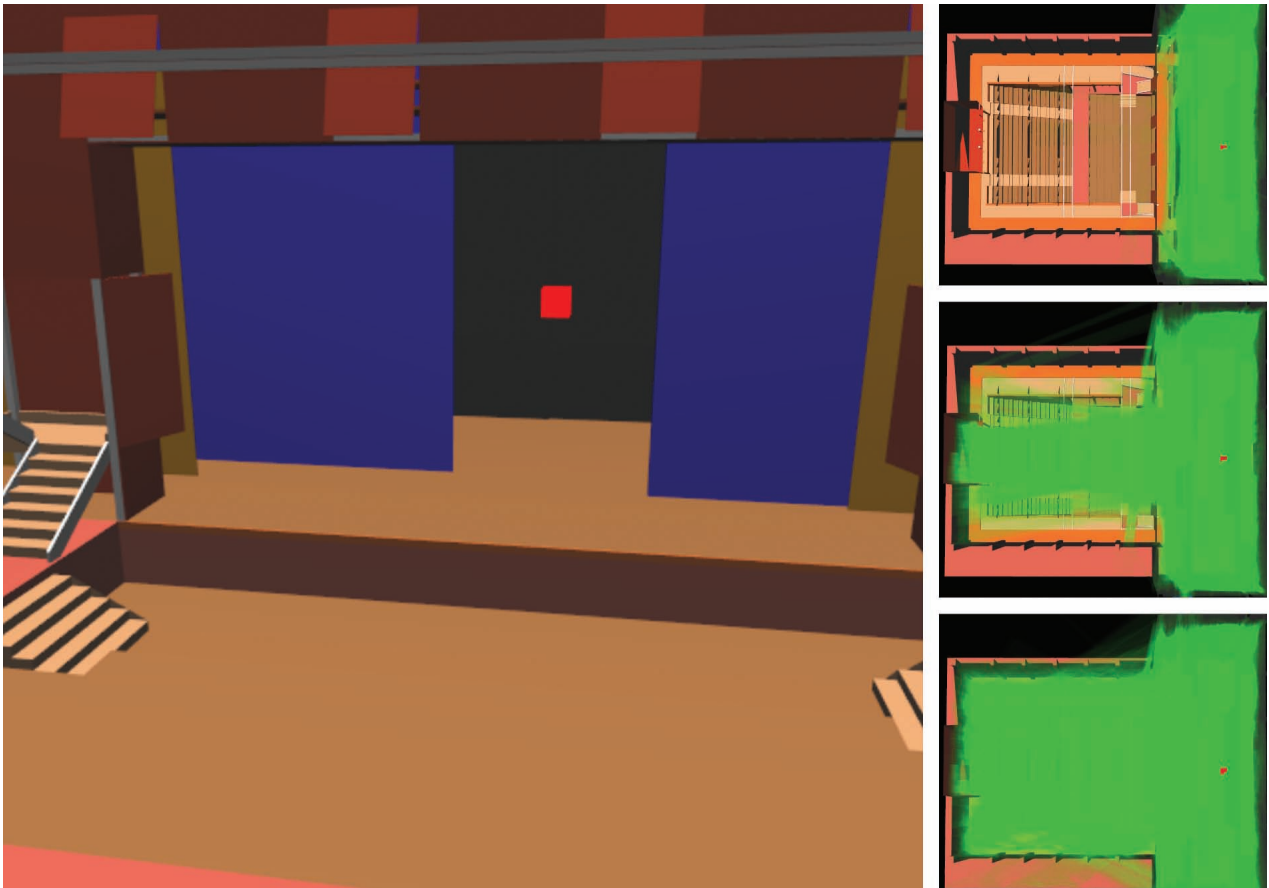


Figure 3. Real-time sound propagation. Screenshots from our system simulating sound propagation. Even though the scene has ~9,000 triangles, the algorithm still computes the sound up to four reflections by shooting more than 95,000 beams more than two times per second on a 2GHz laptop computer. The image sequence, right, shows the computed beams as seen from a top-down view for the first three reflections in a real-time animation of the blue curtains, from fully closed (top) to fully open (bottom); for more video, images, and sounds see gamma.cs.unc.edu/SOUND/.

gamma.cs.unc.edu/SOUND/.

5. Lauterbach, C., Yoon, S.-E., Tuft, D., and Manocha, D. RT-DEFORM: Interactive ray tracing of dynamic scenes using BVHs. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing* (Salt Lake City). IEEE Press, 2006, 39–46.
6. O'Brien, J., Shen, C., and Gatchalian, C. Synthesizing sounds from rigid-body simulations. In the *ACM SIGGRAPH 2002 Symposium on Computer Animation* (San Antonio, TX, July 21–22). ACM Press, New York, 2002, 175–181.
7. Raghuvanshi, N. and Lin, M. Interactive sound synthesis for large-scale environments. In *Proceedings of the ACM Symposium on Interactive 3D Graphics and Games* (Redwood City, CA, Mar. 14–16). ACM Press, New York, 2006, 101–108.
8. Sek, A. and Moore, B. Frequency discrimination as a function of frequency, measured in several ways. *Journal of the Acoustical Society of America* 97, 4 (Apr. 1995), 2479–2486.
9. van den Doel, K., Kry, P., and Pai, D. Foleyautomatic: Physically based sound effects for interactive simulation and animation. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)* (Los Angeles, Aug. 12–17). ACM Press, New York, 2001, 537–544.
10. van den Doel, K. and Pai, D. The sounds of physical shapes. *Presence* 7, 4 (Aug. 1998), 382–395.
11. Wald, I., Boulos, S., and Shirley, P. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics* 26, 1 (Jan. 2007).

NIKUNJ RAGHUVANSHI (nikunj@cs.unc.edu) is a Ph.D. candidate in the Department of Computer Science at the University of North Carolina at Chapel Hill.

CHRISTIAN LAUTERBACH (cl@cs.unc.edu) is a Ph.D. candidate in the Department of Computer Science at the University of North Carolina at Chapel Hill.

ANISH CHANDAK (achandak@cs.unc.edu) is a Ph.D. candidate in the Department of Computer Science at the University of North Carolina at Chapel Hill.

DINESH MANOCHA (dm@cs.unc.edu) is the Phi Delta Theta/Matthew Mason Distinguished Professor of Computer Science at the University of North Carolina at Chapel Hill.

MING C. LIN (lin@cs.unc.edu) is the Beverly W. Long Distinguished Professor of Computer Science at the University of North Carolina at Chapel Hill.

The research described here is supported in part by the National Science Foundation, Office of Naval Research, U.S. Army Research Office, and Intel Corporation. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of the sponsors.

© 2007 ACM 0001-0782/07/0700 \$5.00