Sound Synthesis for Impact Sounds in Video Games

D. Brandon Lloyd   Nikunj Raghuvanshi   Naga K. Govindaraju

eXtreme Computing Group
Microsoft Research

Abstract

We present an interactive system for synthesizing high quality, physically based audio on current video game consoles. From a recorded impact sound, we compute a modal model, which we use to synthesize variations of the sound on the fly. We show that for many sounds greater quality is obtained by using the amplitude envelopes of the extracted modes directly rather than fitting the envelopes to the standard exponential decay model. When combined with a residual, the synthesized sounds in most cases are indistinguishable from recorded clips. Compared to using multiple prerecorded clips to obtain variation, our system consumes less of the limited console memory. For sounds that are less amenable to modal synthesis, we introduce a simple filter that generates plausible variations from a single clip. Our system integrates easily with existing audio middleware and have been implemented in the Xbox360 game *Crackdown II*.

Keywords: sound synthesis, interactive audio

1 Introduction

High quality audio is crucial for achieving realism and creating a compelling user experience in video games. This is especially true when a game features an open world environment where the user can interact with surrounding objects (see Figure 1). For example, the user may throw objects around, stack them up, or knock them over, just as in the real world. When coupled with a physics engine, the animation of the objects can be very realistic. This realism is enhanced when accompanied by realistic audio that corresponds to the animation.

Generating realistic, physically-based audio on today's gaming consoles presents a number of challenges. First, console memory capacities are limited, and audio memory budgets are typically quite low (e.g. 25 MB for *all* audio / 2MB for impact sounds in the game in Figure 1). Prerecorded clips provide high quality contact sounds with very little computation, but they must reside in memory because the latency of streaming from disk is too high. For realistic audio, a single clip for each kind of object is not enough because impact sounds in the real world exhibit small variations in timbre that depend on where the impact occurs on the surface of the object. To achieve sufficient variation, multiple clips are required, which exacerbates the memory problem. Second, the audio system needs to operate within a fixed (usually small) CPU budget. This simplifies the task of distributing computational resources between various subsystems in a game (e.g. graphics, physics, and artificial intelligence). Respecting a fixed budget is a challenge when a large number of impact sounds can be active at any given time. Finally, solutions for realistic audio should ideally integrate easily with existing audio tools and production pipelines, provide sufficient control, and be easy for audio designers to use.

Figure 1: *We generate high-quality sounds in real-time for the open-world video game* Crackdown II *running on an Xbox360 console. The player can interact freely with objects in the environment. Our system synthesizes variations of the resulting impact sounds on the fly with less memory than previous techniques that used multiple prerecorded clips. Our system integrates easily with existing audio middleware, simplifying the authoring process.*

Sound synthesis is an attractive solution to the memory problem. Instead of storing multiple prerecorded clips in memory, sound synthesis can be used to generate variations on the fly. In addition, the underlying synthesis models can often be represented quite compactly. Most existing techniques for interactive, physically-based collision sounds rely on a modal representation that expresses the output sound as a superposition of independently oscillating resonant modes, each with its own characteristic gain, frequency, and exponential decay rate. But the quality of the sounds generated with the idealized model of exponential decay is often unsatisfactory, as we show in our results. Exponential decay can fail to capture the energy transfer between modes [Chadwick et al. 2009] and other nonlinear effects. Even with an accurate physical model, recreating the richness of real world sounds is still challenging because it requires a detailed simulation of the impact forces, which can be difficult to formulate and is beyond the stringent performance requirements of current video games. Modal synthesis also fails to reproduce some sounds that lack strong modal components (e.g. a footstep). There has also been extensive work in musical synthesis to accurately reproduce real sounds. The spectral modeling synthesis (SMS) [Serra and Smith 1990] approach models the frequency spectrum of a sound rather than the underlying physics that generated it. SMS typically decomposes the sound into *partials* - quasi-sinusoidal tracks that vary slowly over time in amplitude and frequency, and a noisy residual. This approach is more general than modal synthesis and can produce higher quality results, but it is also more expensive to compute.

Main Results: In this paper we present a system for synthesizing realistic impact sounds, which was used in the Xbox360 game *Crackdown II*. The system computes variations of the sounds under tight memory and CPU constraints. For modal sounds, we use

I3D 2011, San Francisco, CA, February 18 – 20, 2011.
© 2011 ACM 978-1-4503-0565-5/11/0002 $10.00

modal synthesis, but unlike other interactive modal synthesis techniques, we do not assume ideal exponential decay. Instead we draw on spectral modeling synthesis (SMS) and for each mode we use the actual amplitude envelopes extracted from an impact clip. This approach maintains the good performance of modal synthesis but can greatly improve the quality of the synthesized sounds. In addition we compute a residual to capture details of the sound that are missed in the modal representation. We also handle nonmodal sounds using a multi-dip filter. Our system has three main features:

- Higher quality modal synthesis using sampled mode amplitude envelopes instead of ideal exponential decay
- A low-complexity representation for mode parameters that is both compact and suitable for high performance computation of the synthesized sound
- A filter for nonmodal sounds that can generate variations of a single clip that are similar in character to those generated by modal synthesis

We employ quality scaling techniques that gracefully reduce sound quality in order to meet fixed CPU and memory budgets. Even with a small budget of 10% of a CPU we can handle dozens of simultaneous impact sounds without significant loss of quality. Our system is implemented as a set of plugins for Wwise, a popular audio engine. The plugins are easy to use and require minimal changes to existing audio pipelines.

The rest of this paper is organized as follows. We will first describe how our techniques are related to previous work. In Section 3 we give an overview of our techniques and describe their implementation in Section 4. We then discuss some results in Section 5 and conclude with some ideas about future work.

2 Related Work

The field of sound synthesis using computers is broad. The text by Perry Cook [2002] is a good general introduction to synthesis techniques focussed on interactive applications.

Our approach is closely related to several previous modal synthesis techniques. Van den Doel et al. [2001] use modal models derived from sound samples captured by striking an object at different locations on its surface. The modal model is used to generate impact, rolling, and sliding sounds. We use a single sound sample, typically obtained from a large library of prerecorded sounds. Obrien et al. [2002] compute modal models from a polygonal model and material parameters of an object. Raghuvanshi and Lin [2007] suggest a number of optimizations to reduce the computation needed for real-time modal synthesis. Bonneel et al. [2008] accelerate modal synthesis for real-time applications by combining modes from all active voices in the frequency domain and performing a single inverse Fourier transform to return to the time domain. Much of the efficiency of frequency domain methods is lost when integrating with existing audio middleware, which mixes voices in the time domain and therefore requires an inverse Fourier transform per voice. All of these modal synthesis techniques assume ideal exponential decay for mode amplitudes. Because real world sounds do not always conform exactly to this assumption, the sounds generated by these techniques often sound too ”clean”. Adding arbitrary amplitude envelopes is fairly straightforward for the time domain methods, but is more difficult for frequency domain methods like that of Bonneel et al.

Whether the parameters for the modal model are derived from simulation or from measurement, the intermediate representation is the same. Both approaches have advantages and disadvantages within the context of the audio production pipeline of a video game. The advantage of the simulation approach is that it is possible to create a synthesis model without expensive measurement equipment or the actual physical object. Simulation also computes automatically the variation in impact sound at different locations on the object. One disadvantage of simulation is that it requires a geometric model, which creates additional dependencies between audio production and the rest of the development effort that currently do not exist. Another difficulty is that the material parameters used in the simulation are not necessarily intuitive for audio designers. The main advantage of the data-driven measurement approach is that it is fairly easy to use. The audio designer simply provides an example of the desired result and the system generates a corresponding synthesis model. It also fits well with the typical production pipeline where the audio designer selects sounds from large libraries of prerecorded samples or uses sounds recorded by foley artists.

Our system utilizes aspects of spectral modeling synthesis (SMS) [Serra and Smith 1990]. SMS models a sound as stable sinusoids (partials) plus noise (residual component). Sinusoids are detected in an input sound by analyzing its short-time Fourier transform (STFT). The STFT gives information about the phase and magnitude of the sound over frequency and time. Sinusoids show up as peaks in the magnitude spectrum. Peaks in successive time slices that change relatively slowly are strung together to form partials. Partials are represented as amplitude and frequency samples over time. Subtracting out the partials from the input sound leaves a noisy residual. A coarse spectral envelope is computed from the residual, which is used to shape white noise in the synthesis step.

Our modal synthesis system uses a sampled amplitude envelope like the partials of SMS, but modes have constant frequency, which leads to a number of performance optimizations, which we describe later. We also use a residual, but we do not synthesize it on the fly.

3 Modal synthesis

When an object is struck, the object vibrates at specific frequencies which are called its *resonant modes*. The amplitudes of the mode vibrations decay with time, with low frequency modes generally decaying more slowly than high frequency modes. To reproduce an impact sound, modal synthesis simulates the vibration of the resonant modes. Mathematically, the synthesized signal $x(t)$ can be computed as:

$$x(t) = \sum_{m}^{M} g_m A_m(t) \sin(2\pi f_m(t) + \phi_{0,m}) \; + r(t), \qquad (1)$$

where M is the number of modes and g_m, $A_m(t)$, f_m, and $\phi_{0,m}$ are the gain, amplitude envelope, frequency (Hz), and initial phase of each mode m, respectively. In previous work, the amplitude envelope is assumed to have the form $A_m(t) = e^{-\alpha t}$, where α is the decay constant. This is a good representation for some impact sounds, especially metals. But the amplitude envelopes for recorded sounds are seldom perfectly exponential. We allow $A_m(t)$ to be an arbitrary function, which allows us to support a much broader class of sounds, such as those arising from energy transfer between modes [Chadwick et al. 2009]. We also add a residual term $r(t)$, which can capture components of a sound, such as noise, that are not included in the modal model. Because the combination of arbitrary envelopes and the residual can more faithfully represent the details of the input clip from which they are computed, they can greatly improve the quality of the synthesized sounds.

Because the ideal exponential decay arises from a simplified vibrational (modal) model and assumes a impulsive impact, it fails to capture complex physical interactions during impacts of real world objects.

In this section we describe how we derive the mode parameters by analyzing an input clip and how the synthesis is coupled with a rigid-body physics simulation. We then discuss how to obtain variations of impact sounds using modal synthesis. Finally, we present a number of optimizations that are important for meeting tight performance and memory requirements.

3.1 Analysis

To compute the mode parameters, we analyze the short-time Fourier Transform (STFT), just as in SMS [Serra and Smith 1990]. (The STFT is formed by concatenating the Fourier transforms of a sliding window over the signal.) In a spectrogram (magnitude of the STFT) modes show up as strong peaks that persist over time at the same frequency (e.g. see Figure 2). To identify modes we can identify the strongest peaks (in terms of power) within a user specified time slice of the spectrogram, usually near the onset of the sound where high frequency modes have not fully decayed. Because some peaks may be due to noise, we actually allow the user to select a region covering multiple slices. If the percentage of the slices in which a peak appears is over a given threshold, it is selected as a mode. Once we have identified a mode, we extract its frequency f_m and its amplitude envelope $A_m(t)$, consisting of a sample per time slice. We obtain the residual $r(t)$ as in SMS by synthesizing the modal component and subtracting it from the original signal.

The fidelity of the modal component depends on how well the mode parameters are estimated. Due to intrinsic properties of the STFT, accurate parameter estimation is limited to sinusoids whose frequencies and amplitude change slowly over time and that are not spaced too closely together. Our constant frequency assumption places further restrictions on the sinusoids we can accurately estimate, but leads to a faster implementation. This means that not all impact sounds can be faithfully modeled with modal synthesis. The technique is particularly well suited for sounds with strong resonances.

The performance of the synthesis step depends on the number of modes, so we would like to utilize as few modes as possible. Without a residual, the fidelity also depends on the number of modes used. To enable the user to make a performance/quality trade off, we provide a slider controlling the number of modes to use. Modes are prioritized by their maximum power. Our analysis tool is interactive, so the user can easily optimize the mode count by decreasing slider to just before the point where the quality of the synthesized sound begins to degrade.

The use of the residual introduces several interesting tradeoffs between memory, computation, and quality. The residual often requires more memory than the modal representation but offers the best possible quality. Modes left out of the modal representation fall into the residual, so overall quality is not lost with few modes, but they no longer contribute to the variation of the sound. In addition, the magnitude of the residual grows. If it becomes too large, the residual can drown out the variation in the sound we synthesize from the modal representation. On the other hand, removing modal components from the residual can make it shorter. For sounds with a strong modal component, the energy in the residual is mostly noise concentrated near the attack. The tail is empty and can be clipped. In general, we have found that letting the short duration, high frequency modes fall into the residual keeps the residual short while maintaining fidelity and improving performance.

3.2 Physics engine integration

The physics engine supplies the sound system with impulses arising from object collisions. An impulse is translated into an audio event. A new voice is created and we initialize the mode gains g_m proportional to the impulse value. The sound system then requests audio from the voice, one frame at a time (typically 1024 samples at 48 kHz).

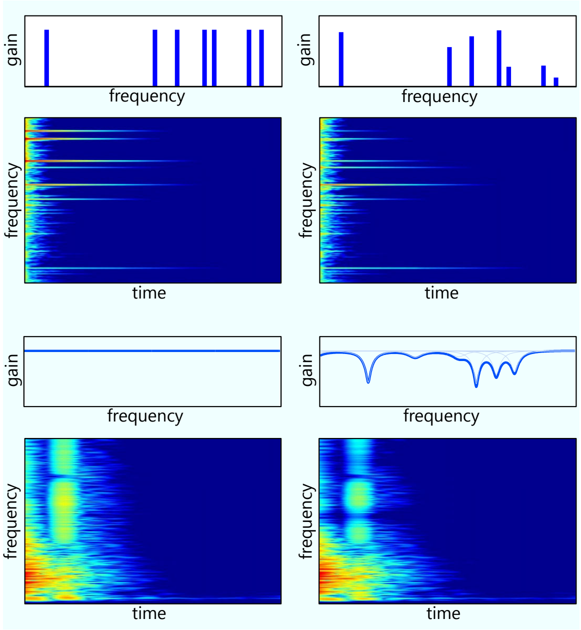

Figure 2: *(Top-left) Mode gains (in dB) and spectrogram of a metallic impact sound. The modes correspond to the streaks in the spectrogram. (Top-right) The mode gains have been randomized to create a variation of the sound. (Bottom-left) The spectral envelope and spectrogram of a footstep sound with a more broadband spectrum. (Bottom-right) A number of randomized dip filters (gray) combine to modify the spectral envelope (blue) to create a variation for the sound.*

Multiple simultaneous impacts can cause problems when using the same prerecorded clip for all impacts. Because the waveforms for the sounds of each impact are all the same, the peaks in the waveforms all line up, and when added together, saturate the dynamic range of the signal. To avoid this problem with modal synthesis we randomize the initial phases $\phi_{0,m}$. We can do this without affecting the sound because, in most cases, a human listener is insensitive to phase. The audio engine coalesces events during one frame and processes them together at the beginning of the next. If the time of the corresponding collision that generated an event is known, the actual onset of the sound can be delayed appropriately. If not, (as in our case) we add a bit of random delay to spread the impacts out slightly in time. This improves the overall sound and can also help somewhat with the clipping problem. Spreading impacts out over multiple frames can also help avoid peaks in computational load [Bonneel et al. 2008].

3.3 Variation

A typical object will sound different depending on where it is struck. The reason for this is that each resonant mode is excited by a different amount depending on the location of impact. Modal models derived from 3D geometry can explicitly account for this spatial variation. In our data-driven approach, we have only one clip and do not know anything about the geometry or material properties of the object that generated the sound, and even if we did, they might not be the same as the object to which the sound is being applied.

We can, however, capture the qualitative aspect of striking an object at random locations by randomizing the mode gains.

The qualitative effect of spatial variation is to attenuate the low frequencies in fine structures or near edges. For example, hitting a box near its edge results in a higher pitched sound than in the center of one of its faces. One possible heuristic approach to capture this effect might be to use a high pass filter on the synthesized sound, with the high pass cutoff encoded in a texture map applied to the object.

3.3.1 Artistic control

In order to give an audio designer artistic control over the amount of variation, we introduce a parameter $v \in [0, 1]$ that varies between no variation ($v = 0$) and full variation ($v = 1$). The actual mode gain g_m that we use for each mode is:

$$g_m(v) = \text{lerp}(1, \xi, \alpha(v))/c(v) \tag{2}$$

$$\text{lerp}(x_0, x_1, \alpha) = x_0 + \alpha(x_1 - x_0) \tag{3}$$

$$\alpha(v) = \text{bias}(v, 0.9) \tag{4}$$

$$c(v) = \sqrt{1 - \alpha(v) + \alpha(v)^2/3} \tag{5}$$

$$\text{bias}(x, b) = \frac{x}{(1/b - 2)(1 - x) + 1} \tag{6}$$

where $\xi \in [0, 1]$ is a uniform random number. The lerp function (Eq. 3) linearly interpolates between x_0 at $\alpha = 0$ and x_1 at $\alpha = 1$. If we use $\alpha = v$ directly, then the perceived change in variation with a change in v is nonlinear. With equal sized steps in v the variation changes much more quickly for v close to 1. To make the v parameter easier to use, we smooth out this nonlinearity using the bias function [Schlick 1994] (Eq. 6). Intuitively speaking, the bias function pushes the value of its argument $x \in [0, 1]$ closer to 1 for $b > 0.5$ and closer to 0 for $b < 0.5$. We found that $b = 0.9$ gives good results. Since average power is proportional to amplitude squared, we normalize average power with the $c(v)$ term, the square root of the expected value of the numerator of $g_m(v)$.

3.4 Optimizations

In order to meet the memory and performance demands of a console game, we use a number of optimizations. To encode the mode parameters, we use a representation that is quite compact, yet simple enough that it can be used directly for high-performance synthesis. We avoid complex compression schemes that would require too much memory or computational overhead for decompression. We also use a quality scaling technique to impose an upper limit on the number of modes that need to be stored and processed. To save computation we mix lower frequency modes at lower rates. Finally, we utilize SIMD vector instructions to speed up the computation of the synthesis equation.

3.4.1 Representation

Using the sampled amplitude envelope rather than fitting it to an exponential decay model results in better quality, but takes up more memory. To reduce the size of the data, we use a combination of simplification, compaction, and quantization for amplitude envelopes.

Simplification. Amplitude envelope samples are stored using a logarithmic scale. To simplify the envelopes we first clamp all values below a user specified noise floor (e.g. -81 dB relative to 0 dB maximum amplitude). Modeling the often erratic amplitude variations below this level does not produce meaningful results. We then simplify the finely sampled envelopes by removing samples one by one until we reach a user specified number of samples. At each step we greedily remove the sample that causes the smallest change in error. The error is measured as the sum of squared differences between samples in the original envelope and corresponding interpolated values of the simplified the envelope. To simplify the representation, we process all the envelopes together so that they share a common set of sample locations. This algorithm is similar to one described by Horner and Beauchamp [Horner and Beauchamp 1996], except that our approach is bottom-up while theirs is top-down. The algorithm is fast enough to be interactive, enabling the user to easily modify the number of samples (using a slider) until the desired quality/size tradeoff is achieved. We note that the samples tend to cluster near the initial attack of the sound where most of the complexity lies rather than in the simpler tail that typically exhibits more or less exponential decay. For an ideal exponential decay, this algorithm simplifies an envelope down to a single line segment. To reproduce the sound most accurately when using simplified envelopes, it is important to linearly interpolate amplitudes in log space. We typically see a 3–4 times reduction in envelope samples from this step.

Compaction. Compaction exploits the fact that many modes die off quickly. Once a mode's amplitude envelope reaches and remains below the noise floor we can truncate it because it will make no audible contribution to the signal. We store only the span of envelope samples for which a mode is active. We pack the spans together into a single array, storing the offset of each span in a table. Compaction commonly yields a 25-50% size reduction.

Quantization. We have found that quantizing the amplitude values to 8-bits yields very little or no perceptual difference in the synthesized sound (loudness discrimination for humans is about 1 dB over all frequencies [Jesteadt et al. 1977]). This gives an additional factor of 4 savings over floating point amplitude values.

The size of the residual can usually be reduced as well. For some sounds, the residual is small enough that it can be left out. As mentioned previously, the tail of the residual can be clipped. Since the residual is also noisy, it can often be heavily compressed without introducing noticeable artifacts. When the residual is small, the modal+residual model can actually be much smaller than the original input.

3.4.2 Quality scaling

The amount of computation required for modal synthesis at any point in time depends on the number of active modes, which is highly variable. For a single impact, the mode count starts out high and then tapers off (see Figure 4). The number of modes also varies depending on the number of objects that are generating sound. Sharp spikes in computational load are undesirable for games. Predictable performance is preferred because it simplifies the task of balancing computational resources. Moreover, if the time to perform the synthesis becomes longer than the time to play a frame of audio, then the audio system will become starved for frames, resulting in jarring clicks and pops. Therefore to ensure that the computation time stays bounded we utilize a mode quota. When we exceed the quota, we prioritize the modes by power and drop lower priority modes. The effect of dropping modes is to gracefully degrade quality and is usually imperceptible. Under extreme conditions (many simultaneous impacts) the typical effect is a reduction in high frequencies, as these tend to have the lowest power. We automatically tune the mode quota to meet a specified CPU usage limit.

An additional benefit of the mode quota is that it puts an upper limit on the memory that will be consumed by modal synthesis voices. Each voice maintains the state for all of its active modes. With a large number of voices, the size of that state could become so large as to erase the memory savings that were one of the motivations for using modal synthesis in the first place. The mode quota provides a solution to this problem by placing tight bounds on memory consumption.

Because our representation ensures that the number of modes decreases monotonically over time, we only need to perform quality scaling when new voices are introduced to the system. We compute the sum S of all active modes over all voices and then reduce the number of modes for each voice by the fraction $(S - B)/S$, where B is the mode budget. Better results may be achieved by removing the $(S - B)$ lowest priority modes instead of removing a fixed fraction of modes from all active voices, but we have not yet tried this. We would also like to try clustering voices as in Tsingos et al. [2004].

3.4.3 Multirate mixing

To accelerate modal synthesis we mix lower frequency modes at a lower rate [Phillips 1999]. We first divide the modes into frequency bands (e.g. Figure 4). We mix the bands from lowest to highest frequency, using a $2\times$ interpolation filter to convert the sampling rate from one band to the next. The interpolation filter slightly increases the complexity of the mixing because it requires values beyond the edge of the frame (we use an 8-tap filter). Multirate mixing adds a small constant overhead for rate conversion between bands, but for sounds that have a large number of low frequency modes, which is generally the case for impact sounds, the savings can be significant. For sounds having few low frequency modes, single rate mode mixing should be used to avoid the overhead.

When the span of amplitude data for a mode comes to an end, the mode is terminated. Terminated modes are removed at the end of a frame and the oscillator array is compacted. Oscillators are stored in order of frequency to facilitate multirate mixing. For quality scaling, it is necessary to remove the nodes in order of priority. Therefore we maintain linked list of modes in priority order, which is also compacted when terminated modes are removed.

3.4.4 SIMD optimization

Mode mixing is a highly data-parallel operation that lends itself well to optimization on the GPU or a SIMD instruction set. We have explored both options but focus on SIMD optimizations in this paper. SIMD instruction sets typically lack a fast vector $\sin()$ function. But since mode frequency is constant, we can use complex harmonic oscillators to generate the sinusoids for modal synthesis. The $\sin()$ function of Equation 1 for mode m and sample k is computed as $\mathrm{Im}\{c_m^{(k)}\}$, where $c_m^{(k)} = c_m^{(k-1)} \Delta c_m$, $\Delta c_m = e^{2\pi i f_m / f_s}$, $c_m^{(0)} = \Delta c_m e^{\phi_{0,m}}$, and f_s is the sampling rate (typically 48000 Hz). The multiplier Δc_m for each oscillator c_m can be precomputed and shared between voices to save memory.

We process modes in blocks. For each block, we store oscillators and their amplitudes (premultiplied by the mode gains) in vector registers. The real and imaginary components of complex values are stored in separate registers that are four floats wide. At each step we convert log scale amplitudes to linear scale, multiply them by the corresponding imaginary component vectors of the oscillators, sum the results into a single vector register, and take a dot product with $(1, 1, 1, 1)$ to get a single scalar sample value. This value is then added into the audio frame buffer. Then we update the amplitudes and oscillators and repeat for the rest of the samples.

Floating point vector instructions are usually pipelined. When the block size is small, instruction dependency stalls are more likely to occur, decreasing efficiency. For smaller block sizes, we compute multiple samples per step to increase the available instruction parallelism and avoid stalls. We do this by recursively doubling the increment values until we reach the desired power-of-two step size. Efficiency with small block sizes is especially important with multirate mixing because band boundaries fragment large mode blocks into smaller ones.

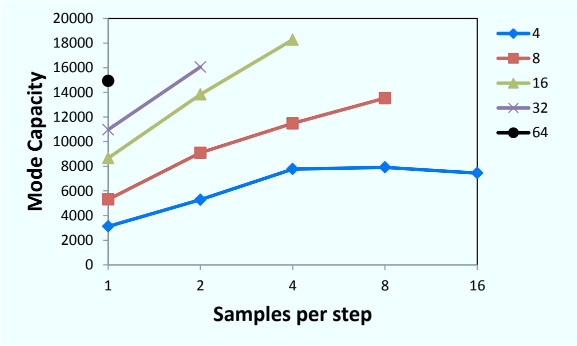

Figure 3: *Number of modes that can be handled with a frame size of 1024 samples and a 48 kHz sampling rate for a varying number of modes per block.*

4 Variation for nonmodal sounds

We would like to be able to apply variation to sounds that are not represented well by modal synthesis. Based on the observation that convincing variation can be generated by randomizing mode gains, we developed a simple filter that attenuates random portions of the frequency spectrum. This can be viewed as a type of subtractive synthesis [Cook 2002]. The filter can produce natural sounding variation for many sounds.

Our variation filter is a cascade of biquad dip (or cut) filters [Zölzer 1997] (see Figure 2). The user specifies the number of dip filters to use. This is analogous to the number of modes for modal synthesis, except that each filter affects a broader range of frequencies. We have obtained good results with about 10 filters. The filters are centered at random frequencies (on the log frequency scale). The user also specifies a range for a random filter gain and the a random Q-factor. The Q-factor is related to the width of the dip.

The variation filter is complementary to modal synthesis. While randomized mode gains alter the frequency at specific frequencies where most of the power is concentrated, the variation filter affects broader swaths of the frequency spectrum. The sounds for which the variation filter seems to work best have broadband spectra that are difficult to represent with a collection of narrow band modes (like “Rock” in Figure 5).

Quality scaling can be applied to the variation filter to limit memory and computation usage by scaling back the number of dip filters used.

SIMD optimization. We vectorize the filters by processing 4 samples at a time. As with modal synthesis, we process blocks of dip filters at a time. With too few filters, we run into problems with dependency stalls. To ameliorate this problem, the recursive filter can be unrolled to generate more instructions that can be run in between dependent instructions [Trebien and Oliveira 2009] in order to eliminate pipeline stalls, but this comes at the expense of more operations per sample.

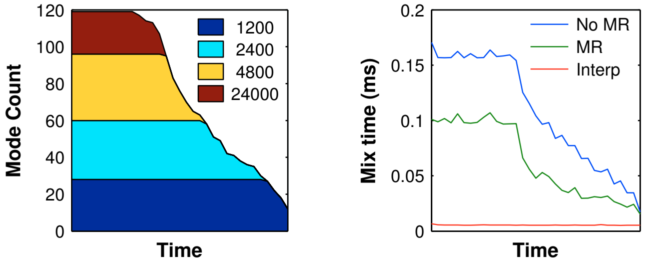

Figure 4: *(Left)The number of active modes per band over the duration of an impact sound at 48000 Hz sampling rate. The colors correspond to the band cutoff frequency in Hz. (Right) The compute time with (*MR*) and without (*No MR*) multirate mixing. The overhead for interpolating between bands (*Interp*) is also shown. For this sound, multirate mixing gives up to a* $2\times$ *speed up.*

Sound	Original Size	Modal Size	Mode Count	Residual Size
Brass Bell	550K(18.5K)	492	6	65.2K(3.2K)
Plastic Barrel	141K(2.4K)	998	23	28.7K(1.7K)
Wooden Box	14K(2.2K)	2381	65	21.1K(1.7K)
Rock	205K(1.6K)	4064	120	11.9K(1.1K)

Figure 5: *This table shows the sizes in bytes of the original input file, the modal representation, and the residual for a number of different objects. The compressed size of the original and residual are also shown in parentheses. For comparison, both the original and the residual were compressed with XMA2 using the maximum possible compression. While the maximum compression is sufficient for the residual, a lower setting, and thus a larger size, is required to get satisfactory quality for the original clip.*

5 Implementation and results

We implemented our techniques on the Xbox360 game console. We perform all of our computation using the CPU of the Xbox360. The GPU is likely a good candidate for some of these computations [Zhang et al. 2005; Trebien and Oliveira 2009], but we have not yet pursued this option. The Xbox360 has 3 Xenon CPUs running at 3.2 GHz, each with 2 hardware threads. The CPUs use a modification of the VMX (Altivec) instruction set with 4-wide SIMD vectors and 128 vector registers. There are two sets of vector pipelines, one for simple operations, such as permutations, loads, and stores, and another for more complex operations like floating point math and dot products. Because instructions are processed in pairs, it is possible for some vector instructions to coissue. We make heavy use of vector intrinsics for our implementation, relying on the compiler to optimize instruction scheduling. We have found that the current compiler (XDK 11164.1) sometimes generates unnecessary dependency stalls and sometimes fails to fully exploit coissue. Coding in assembly language would correct these problems, but we have not yet done this.

As described in Section 3, we interpolate amplitude envelope data in the logarithmic space. This requires computing an exponential per sample to convert to linear magnitude. Fortunately, this adds very little to overall run time because the Xbox360 has an exponential estimate instruction with sufficient accuracy for audio (16-bits).

Figure 3 shows the efficiency of our vectorized implementation of Equation 1 for various block configurations. The large number of vector registers on the Xbox360 allows us to perform the computations completely in registers for up to 64 modes. Figure 4 shows the performance of multirate mixing. We see typical speed ups of up to 2 times. For the sounds that we use, we have found that the greatest savings are obtained with 3–4 bands.

We have implemented modal synthesis and the variation filter as plugins for the Wwise audio engine. In this way, our algorithms can be incorporated directly into an existing pipeline without modification.

The accompanying video contains a number of examples of the application of modal synthesis and the variation filter to various sound clips. We also demonstrate the use of our system in a modified version of the game *Crackdown II*. For several objects, we have either replaced the sound clips that the artist originally used with a modal synthesis plugin or we have applied the variation filter, resulting in a reduction from the typical 3–5 clips used for each sound down to 1 or 2. For some sounds, the modal synthesis+residual model can yield additional memory savings (see Figure 5). This is especially true of metallic objects with a long ring (e.g. a bell). Clips on the Xbox360 are usually compressed with XMA, a variant of WMA, because there is a XMA decoder in hardware. After XMA compression, most of the memory savings with our method comes from a reduced number of clips rather than the size of the modal representation itself.

5.1 Limitations

Arbitrary amplitude envelopes generate better sounds than the ideal exponential decay model, but they do come with a modest cost. The mode amplitude has to be tracked separately from the sinusoid, rather than being folded into the complex oscillator as with exponential decay.

Because our current variation techniques only modify the frequency content of a sound, they are sometimes unsatisfactory for impact sounds composed of many smaller impacts or fractures. Our methods produce no variation in the timing between these "microevents", which can sound unnatural. Granular synthesis is a promising avenue for generating these kinds of variations [Picard et al. 2009].

Our current preprocessing tool has a number of parameters. To relieve the burden on the user we would like to further automate the tool. For example, by introducing a perceptual quality metric, it should be possible to compute an initial set of parameters that are close to optimal. The user could then make a few small tweaks if desired.

6 Conclusion

We have presented a system for synthesizing variations of impact sounds in real-time. We use both modal synthesis and a simple, randomized filter. Our methods enable us to save memory on gaming consoles and meet the performance requirements of a game. We demonstrate the system in an actual video game. We have found that arbitrary amplitude envelopes extracted from actual recordings can greatly improve the quality of modal synthesis over an ideal exponential decay.

References

BONNEEL, N., DRETTAKIS, G., TSINGOS, N., DELMON, I. V., AND JAMES, D. 2008. Fast modal sounds with scalable frequency-domain synthesis. *ACM Transactions on Graphics (SIGGRAPH Conference Proceedings) 27*, 3.

CHADWICK, J. N., AN, S. S., AND JAMES, D. L. 2009. Harmonic shells: a practical nonlinear sound model for near-rigid thin shells. In *SIGGRAPH Asia '09: ACM SIGGRAPH Asia 2009 papers*, ACM, New York, NY, USA, 1–10.

COOK, P. R. 2002. *Real Sound Synthesis for Interactive Applications (Book & CD-ROM)*, 1st ed. AK Peters, Ltd.

HORNER, A., AND BEAUCHAMP, J. 1996. Piecewise-linear approximation of additive synthesis envelopes: A comparison of various methods. *Computer Music Journal 20*, 2, 72–95.

JESTEADT, W., WIER, C. C., AND GREEN, D. M. 1977. Intensity discrimination as a function of frequency and sensation level. *The Journal of the Acoustical Society of America 61*, 1, 169–177.

O'BRIEN, J. F., SHEN, C., AND GATCHALIAN, C. M. 2002. Synthesizing sounds from rigid-body simulations. In *SCA '02: Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, ACM, New York, NY, USA, 175–181.

PHILLIPS, D. K. 1999. Multirate additive synthesis. *Computer Music Journal 23*, 1, 28–40.

PICARD, C., TSINGOS, N., AND FAURE, F. 2009. Retargetting example sounds to interactive physics-driven animations. In *AES 35th International Conference-Audio for Games, London, UK*.

RAGHUVANSHI, N., AND LIN, M. C. 2007. Physically based sound synthesis for large-scale virtual environments. *IEEE Computer Graphics and Applications 27*, 1, 14–18.

SCHLICK, C. 1994. Fast alternatives to Perlin's bias and gain functions. In *Graphics Gems IV*. Academic Press Professional, Inc., San Diego, CA, USA, 401–403.

SERRA, X., AND SMITH, J. 1990. Spectral modeling synthesis a sound analysis/synthesis based on a deterministic plus stochastic decomposition. *Computer Music Journal 14*, 12–24. SMS.

TREBIEN, F., AND OLIVEIRA, M. 2009. Realistic real-time sound re-synthesis and processing for interactive virtual worlds. *The Visual Computer 25*, 5 (May), 469–477.

TSINGOS, N., GALLO, E., AND DRETTAKIS, G. 2004. Perceptual audio rendering of complex virtual environments. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, ACM, New York, NY, USA, vol. 23, 249–258.

VAN DEN DOEL, K., KRY, P. G., AND PAI, D. K. 2001. FoleyAutomatic: physically-based sound effects for interactive simulation and animation. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 537–544.

ZHANG, Q., YE, L., AND PAN, Z. 2005. Physically-based sound synthesis on GPUs. In *Entertainment Computing - ICEC 2005*, F. Kishino, Y. Kitamura, H. Kato, and N. Nagata, Eds., vol. 3711. Springer Berlin Heidelberg, Berlin, Heidelberg, ch. 32, 328–333.

ZÖLZER, U. 1997. *Digital Audio Signal Processing*. John Wiley & Sons Software.