Modal Synthesis for Vibrating Objects *

Kees van den Doel and Dinesh K. Pai
Department of Computer Science
University of British Columbia
Vancouver, Canada
{kvdoel | pai}@cs.ubc.ca

1 Introduction

When a solid object is struck, scraped, or engages in other external interactions, the forces at the contact point causes deformations to propagate through the body, causing its outer surfaces to vibrate and emit sound waves. Examples of musical instruments utilizing solid objects like this are the marimba, the xylophone, and bells.

The sounds made by objects like this are important for interacting with our environment because they provide useful information about the physical attributes of the object, its environment, and the contact events, including the force (or energy) of the impact, the material composition of the object, the shape and size, the place of impact on the object, and finally the location and environment of the object.

In order to create the sounds of objects like this in an interactive digital environment, such as a video game or a simulation, we need real-time synthesis, as we do not know the stimulus of the (virtual) objects before they occur, and sustained intimate user interaction like touching and scraping an object needs a continuously parametrizable sound.

A good physically motivated synthesis model for objects like this is modal synthesis (Wawrzynek, 1989; Gaver, 1993; Morrison & Adrien, 1993; Cook, 1996; Doel & Pai, 1996; Doel, Kry, & Pai, 2001; O'Brien, Chen, & Gatchalian, 2002; Doel, Pai, Adam, Kortchmar, & Pichora-Fuller, 2002), where a vibrating object is modeled by a bank of damped harmonic oscillators which are excited by an external stimulus. The frequencies and dampings of the oscillators are determined by the geometry and material properties (such as elasticity) of the object and the coupling gains are determined by the location of the force applied to the object.

The modal synthesis model is physically well motivated, as the linear partial differential equation for a vibrating system, with appropriate boundary conditions, has as solutions a superposition of vibration modes. See Fig. 1 for an illustration.

Modal synthesis can also be used to model other types of physical systems which can be modeled by excitations acting on resonances, such as car engines, rumbling sounds, or virtual musical instruments. For musical instruments with a harmonic spectrum modal synthesis can be used, but it is computationally quite demanding because of the large number of modes needed. Waveguide models (Smith, 1992; Cook, 2003) are in most cases much more efficient for these types of sounds. The sound made by a modal model can be computed very efficiently with an $O(N)$ algorithm (Gaver, 1993; Doel & Pai, 1998; Doel, 1998) for a model of N modes, as described below.

*This work was supported in part by grants from the Peter Wall Institute and the Institute for Robotics and Intelligent Systems.

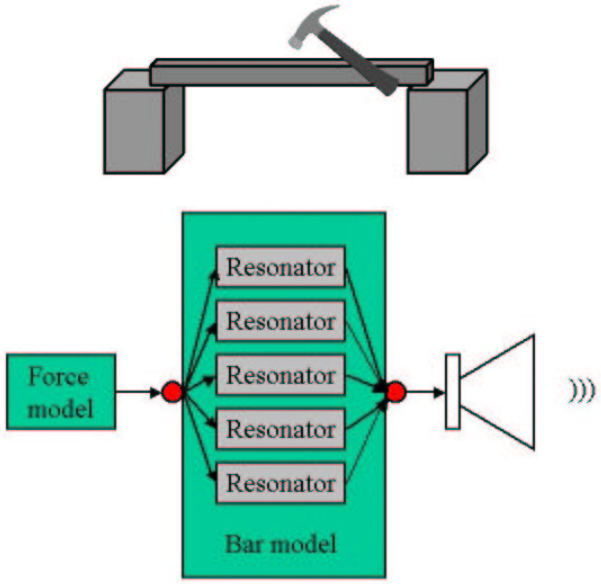

Figure 1: Modal synthesis of the sound made by hitting a bar with a hammer. The hammer force is modeled by a contact force model, and send to a bank of resonators, which is the modal model of the bar. Each resonator has a characteristic frequency, damping, and gain and the outputs of the resonators are summed and rendered.

The remainder of this article is organized as follows. Section 2 defines modal synthesis and explains the physical motivation behind it. An efficient synthesis algorithm is derived formally and we show how to implement it. In Section 3 we discuss the construction of excitation signals to the modal models for a number of applications. Concluding remarks are presented in Section 4. A set of Java classes implementing these ideas is provided on the accompanying CD, and discussed in detail in this text. An implementation in C is also provided.

2 Modal Resonance Models

2.1 General Properties

We can formally describe a modal model $\mathcal{M}$ utilizing N modes at K different locations (contact points) on an object as $\mathcal{M} = \{\boldsymbol{f}, \boldsymbol{d}, \boldsymbol{A}\}$, where $\boldsymbol{f}$ is a vector of length N whose components are the modal frequencies in Hertz, $\boldsymbol{d}$ is a vector of length N whose components are the (angular) decay rates in Hertz, and $\boldsymbol{A}$ is an $N \times K$ matrix , whose

elements a_{nk} are the gain coefficients for each mode.

The Java class **ModalModel** encapsulates a modal model. It contains the public member variables

```
public double[] f;
public double[] d;
public double[][] a;
```

which define $\mathcal{M}$. This class also contains the member variables

```
public double fscale;
public double dscale;
public double ascale;
```

which provide a convenient means to uniformly scale the frequencies, dampings, and gains of an entire model. The class constructor `ModalModel(String fn)` reads the modal parameters from a text file in a self-explanatory format. By convention a modes file has the extension `.sy`. Several examples of modal models are provided with the accompanying code examples.

The impulse response $y(t)$ of $\mathcal{M}$ at location k is given by

$$y(t) = \sum_{n=1}^{N} a_{nk} \exp(-d_n t) \sin(2\pi f_n t), \tag{1}$$

for $t \geq 0$ and is zero for $t < 0$, where $y(t)$ denotes the audio signal as a function of time. The impulse response represents the sound the virtual object makes when struck with a unit impulse at time $t = 0$ at location k. The decay rate d_n of each mode is an object property which is strongly influenced by the material, which determines the internal dissipation of energy during vibration. According to a simple material model explained in (Wildes & Richards, 1988; Krotkov & Klatzky, 1995), the dampings d_n are just proportional to the modal frequencies f_n, i.e., $d_n = \rho f_n$, with the proportionality constant ρ determined by the internal friction parameter. Small values of ρ produce models which are relatively undamped, characteristic of metal objects, whereas larger values produce highly damped models characteristic of materials such as plastic and wood. In real objects this relation between damping and frequency is only approximately valid (Doel, 1998), but this simple model is capable of evoking the illusion of materials reasonably well as was shown in perception studies (Klatzky, Pai, & Krotkov, 2000). The impulse response y as given in Equation 1 is characteristic of physical systems that obey the linear wave equation for solid bodies, which is of the form

$$(A - \frac{1}{c^2}\frac{\partial^2}{\partial t^2})\mu(\boldsymbol{x}, t) = 0 \tag{2}$$

on some domain, where μ is the deviation of the surface as a function of time, and A is a (usually very complicated) spatial differential operator. The solution of Equation 2, together with a radiation model which we do not discuss here, in principle allows the calculation of the modal parameters in Equation 1, however this is very complicated. In (Doel & Pai, 1998) the calculation was performed for some simple shapes. Finite element methods were used in (O'Brien, Cook, & Essl, 2001; O'Brien et al., 2002).

The provided modes file `s100.sy` contains the computed modal data for an ideal string, with 100 modes and 20 contact locations. Taking the string to lie on the interval [0 1], the computed contact locations are at the K discrete points $p = (k + 1)/2K$, where $k = 0, \ldots, K - 1$. The demo **DemoBowedString** uses this modes file to synthesize an ideal string bowed by white noise. You can set the bow point with the bottom slider whereas the upper slider sets the pitch (or string tension). A linear model with a noise excitation oversimplies the physics of a bowed string, which is non-linear and quite complicated (Serafin, 2003). Nevertheless, the resulting sound is quite convincing.

The sound model parameters for a given object can also be obtained experimentally by recording the impulse response of the object and fitting the model parameters to the recorded sound. We can think of this as designing a digital filter of a specific type with a given impulse response (the recording). Various off-the-shelf tools are available to display spectrograms and sonograms of sounds, and these can be used to measure the modal parameters. For example, in Fig. 2 we depict the spectrogram of a recording of

Figure 2: The spectrogram of a church bell. The x-axis is time, frequency corresponds to the y-axis. The decibel level is mapped to darkness.

a church bell (`bell4.wav`). An algorithm to automatically extract a modal model from a recorded impulse response at one location was given in Van den Doel's thesis (Doel, 1998). A modified version of the algorithm was used to extract the modes of the bell which can be found in the file `bell4.sy`. This algorithm is also capable of integrating data from multiple contact points into a single modal model and is described in (Pai et al., 2001). The corresponding demo is **DemoBellStrike**. The lower slider in the UI dialog sets the number of modes used in the synthesis, from $1 - 50$, the upper slider sets the hardness (see Section 3.1) of the virtual mallet used to strike the bell.

The Active Measurement Facility (ACME) at the University of British Columbia (Pai et al., 2001) has the capability to automatically acquire sound measurements by moving a sound effector around the surface of a test object by the robot arm. At selected points on the surface, the sound effector hits the object with an impulsive force and records the sound produced by the impact. The modes file `calona0.sy` contains the modes extracted from measurements at ACME on a glass bottle. The corresponding demo is **DemoBottleHit**. The lower slider in the control panel sets the location of the impact point on the bottle, mapped to the interval [0 1], and the upper slider sets the hardness (see Section 3.1) of the virtual mallet used to strike the bell.

Sometimes it is desirable to construct a modal model by hand (and ear), for example using a modal model editor (Chaudhary, Freed, Khoury, & Wessel, 1998). An example of a resonance model for engine sounds is given in `car1.sy`, which is used in the **DemoEngine** demo, which takes command line arguments pointing to the modal model

and the excitation model. We will discuss this demo further in Section 3.3.

2.2 Derivation of a Modal Synthesis Algorithm

We shall now derive the modal synthesis algorithm, show how it can be implemented most efficiently, and we then show that it is in fact a bank of reson filters (resonant bandpass filters) operating on the interaction force. If we assume a linear model, the response to any kind of input force is determined completely by the impulse response. It follows from Equation 1 that the sound produced by an impulsive force of magnitude F at time s can be described by the imaginary part of the complex wave form

$$y(t) = \sum_n a_n e^{i\Omega_n(t-s)} H(t-s)F, \tag{3}$$

where the sum is over the complex eigenfrequencies Ω_n (the imaginary part determines the damping of a mode). $H(t) = 0$ for $t < 0$ and $H(t) = 1$ for $t \geq 0$. If we substitute $F = 1$, $s = 0$, $\Omega = 2\pi f_n - id_n$, assume $t > s$ so $H(t-s) = 1$, we recover Equation 1 from this.

A continuous stimulus force $F(t)$ can be represented formally as an infinite sum of infinitesimal impulses

$$F(t) = \int_0^\infty \delta(t-s)F(s)ds,$$

where $\delta(t)$ is the Dirac delta distribution, assuming the force is zero for negative times. Using the principle of linearity, the output of the model driven by this force can be written as a sum of infinitesimal contributions from each of these impulses:

$$y(t) = \int_0^\infty ds \sum_n a_n e^{i\Omega_n(t-s)} H(t-s)F(s).$$

Discretizing this equation in time, with sampling rate S_R, gives

$$y(m) = \sum_{l=0}^{m} \sum_n a_n e^{i\frac{\Omega_n}{S_R}(m-l)} F(l),$$

with $y(m) = S_R y(t_m)$ and $t_m = m/S_R$. This convolution equation can be rewritten as a recursion relation by defining the functions $y_n(m)$, one for each partial. The complex signal is written as a sum of modal contributions y_n

$$y(m) = \sum_n y_n(m).$$

For the partials $y_n(m)$ we have

$$y_n(0) = a_n F(0)$$

and the recursion relation

$$y_n(m) = e^{i\frac{\Omega_n}{S_R}} y_n(m-1) + a_n F(m) \tag{4}$$

determines the audio signal $\mathrm{Im}(y)$. As $|e^{i\frac{\Omega_n}{S_R}}| < 1$, the recursion relation is always stable. Equation 4 requires 5 multiplications per sample point, which can be reduced to 3 as we will now derive.

To simplify the notation, let us drop the subscripts n which labels the modes, and write $y(m) = u(m) + iv(m)$, with u and v real. The recursion can now be written as

$$\begin{aligned} u(m) &= c_r u(m-1) - c_i v(m-1) + aF(m) \\ v(m) &= c_i u(m-1) + c_r v(m-1), \end{aligned} \tag{5}$$

with

$$c_r = e^{-d/S_R}\cos(\omega/S_R),$$
$$c_i = e^{-d/S_R}\sin(\omega/S_R),$$
$$d = Im(\Omega),$$

and

$$\omega = Re(\Omega).$$

We can eliminate u from Equation 5 as

$$u(m) = v(m+1)/c_i - c_r v(m)/c_i$$

and arrive at the second order recursion for the quantity of interest, v, in the form

$$v(m) = 2R\cos(\theta)v(m-1) - R^2 v(m-2) + aR\sin(\theta)F(m-1), \tag{6}$$

$$R = e^{-d/S_R},$$

and

$$\theta = \omega/S_R.$$

Equation 6 is precisely the equation for a reson filter (Steiglitz, 1996) with transfer function $\mathcal{H}(z) = 1/(1 - 2R\cos\theta z^{-1} + R^2 z^{-2})$, operating on the input signal $aR\sin(\theta)z^{-1}F$. Note that because of the Nyquist theorem all modal frequencies must be less than half the sampling rate, so both $\sin > 0$ and $\cos > 0$.

To synthesize the sound in real-time, we repeatedly compute an audio buffer of length T. The synthesis algorithm fetches the values of the coefficient arrays as well as the external force F for the time interval T. Equation 6 is then used to sequentially add contributions of the modes v_n until all modes have been added or until a certain deadline has been passed. Note that this can't be done in-place and requires accumulating intermediate results in separate buffers. If the modes are sorted in a decreasing order of importance, this allows for a graceful degradation in the quality of the synthesized sound, when the time available for audio synthesis is not constant.

2.3 Implementation of Real-time Algorithm

A number of Java classes are provided which implement audio synthesis based on the algorithm presented mathematically in Section 2.2.

The **AudioForce** interface provides a single method which fills a given buffer with samples of the output produced by the object implementing it:

```
public interface AudioForce {
    public void getForce(double [] output, int nsamples);
}
```

Stimulus forces will be represented by objects implementing **AudioForce**, but vibrating objects also implement this interface, in which case they can send their output (obtained through `getForce()`) directly to the audio hardware; they apply their "audio force" to the air. Objects can also excite each other, the output of one becoming the input of the other. This "patch-based" design has been used for a long time in computer music (Mathews, 1969).

An object which produces sound is represented by the abstract base class **SonicObject**:

```
public abstract class SonicObject
          extends Thread implements AudioForce
```

A **SonicObject** is capable of rendering itself to the audio hardware in real-time by creating a thread using the `start()` method which it inherits from its superclass **Thread**, which will call the `run()` method. Its `run()` method,

```
public void run() {
    float [] y = new float[bufferSize];
    RTPlay pb = new RTPlay(bufferSizeJavaSound,srate,16,1,true);
    while(true) {
        this.getForce(y,bufferSize);
        // Unimportant code omitted ...
        pb.write(y);
    }
}
```

obtains the output of the **SonicObject** using its own **AudioForce** interface which classes derived from it must implement, and renders the result using the utility class **RTPlay** which we wrote to provide a convenient interface to the standard Java audio API (JavaSound, which is part of Java 2) through its `write()` method which places a buffer on the playback queue and blocks if the queue is full. This design is illustrated in Fig. 3.

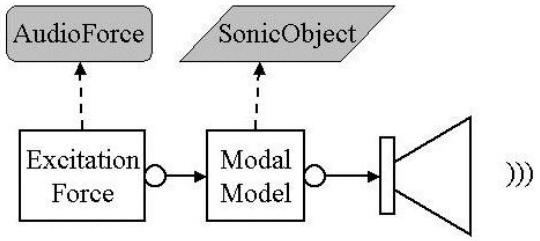

Figure 3: The model for the excitation implements the **AudioForce** interface. It sends audio buffers to the modal model, which derives from the abstract class **SonicObject**, which also implements **AudioForce**.

The Java class **ModalSonicObject** derives from **SonicObject** and implements the modal synthesis algorithm from the previous section. It contains a reference to a **ModalModel**, and a reference to an object `audioForce` which implements the **AudioForce** interface method and which encapsulates the input to the modal resonators. We call it "audioForce" to stress its interpretation as a rapidly fluctuating physical contact force on a material object.

ModalSonicObject implements its own **AudioForce** interface with:

```
public void getForce(float [] output, int nsamples) {
    audioForce.getForce(scratchBuf, nsamples);
    computeSoundBuffer(output, scratchBuf, nsamples);
}
```

Its core method is `computeSoundBuffer()`:

```
private void computeSoundBuffer(float[] output,
                  float[] force, int nsamples) {
    for(int k=0;k<nsamples;k++) {
        output[k] = 0;
    }
    int nf = modalModel.nfUsed;
    for(int i=0;i<nf;i++) {
        float tmp_twoRCosTheta = twoRCosTheta[i];
        float tmp_R2 = R2[i];
        float tmp_a = ampR[i];
        float tmp_yt_1 = yt_1[i];
        float tmp_yt_2 = yt_2[i];
        for(int k=0;k<nsamples;k++) {
            float ynew = tmp_twoRCosTheta * tmp_yt_1 -
                tmp_R2 * tmp_yt_2 + tmp_a * force[k];
            tmp_yt_2 = tmp_yt_1;
            tmp_yt_1 = ynew;
            output[k] += ynew;
        }
        yt_1[i] = tmp_yt_1;
        yt_2[i] = tmp_yt_2;
    }
}
```

The bank of `nf` reson filters has filter coefficients `twoRCosTheta` and `R2`, which correspond to the variables occurring in Equation 6, and gains `ampR`. In the outer `for` loop the filter coefficients are stored in temporary variables to avoid array access in the inner loop. The inner `for` loop adds the contribution of reson `i` to the `output`, using the input buffer `force`. The member variables `yt_1` and `yt_2` remember the last two outputs from one buffer to the next. (Note that we have ignored the one sample delay in the force occurring in Equation 6, as it has no audible effect as long as we don't create any feedback loops using this filter.) Note that the factor $R\sin(\theta)$, which multiplies the input force in Equation 1, has been absorbed in the rescaled gains `ampR`.

There are two types of buffer sizes that concern us here. The first is the buffer size `bufferSize` used in calls to `computeSoundBuffer()`, which has an effect on the latency. It should be set to a low value. However, a low value will introduce more overhead in calls to `computeSoundBuffer()`. The demo class **DemoModalBenchmark** can be used to monitor the performance of the algorithm by varying various parameters like `bufferSize`. It times `computeSoundBuffer()` and then computes how many modes could be synthesized maximally in real-time on the machine. On a $450Mhz$ Pentium III one can synthesize 800 modes at a sampling rate of $22050Hz$, for a buffer size of 128. If we set `bufferSize` to 1 (which will give the lowest latency but the highest overhead) we can do only 200 modes.

In the C sample code directory we also provide a similar benchmark implemented in C. For a buffer size of 128 it was found that we could synthesize 1000 modes, making the C code about 25% more efficient.

The second buffer size is `bufferSizeJavaSound` which should be set to the smallest possible value to obtain the lowest possible latency. The current JavaSound implementations require this buffer size to be rather enormous on most platforms, with very large latencies, but this may improve in the future.

Real-time interaction with a running **SonicObject** is achieved by changing the modal model parameters and/or the contact location. The modal parameters can be accessed directly. Once the **SonicObject** is running, changing the modal parameters has no effect until `computeFilter()` is called, which then computes the filter coefficients occurring in the synthesis loop. There is a potential race condition here, as `computeSoundBuffer()` may be accessing these variables in the synthesis thread whereas a control thread is changing them by calling `computeFilter()`. We have not encountered any such problems in practice, and therefore have not bothered writing the appropriate `synchronized` accessors.

The manner in which the location on the object is set is somewhat involved. The modal model $\mathcal{M}$ contains gains on a discrete set of location points. These location points will usually correspond to geometric locations on some two-dimensional contact surface. In order to allow for interpola-

tion in between the data points we specify the location by providing three discrete location points `p1 p2 p3`, which index the gain arrays. We then draw an imaginary triangle (see Fig. 4) between the physical geometrical points on our

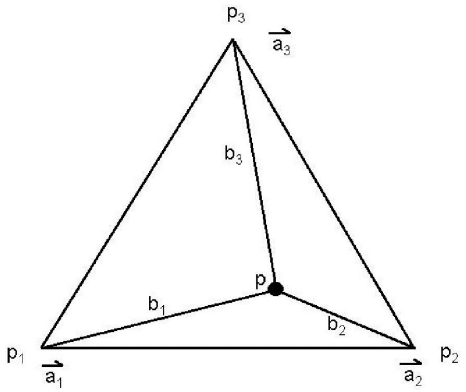

Figure 4: Interpolation of gain vectors which are defined at the corners of the triangle. The gain vector associated with point **p** inside the triangle is the linear combination with the three barycentric coordinates b_i as weights.

actual object (of which the audio objects have no knowledge) and provide three barycentric coordinates `b1 b2 b3` to specify the position within this triangle. The gain coefficients are then appropriately averaged over the three discrete data points. For one-dimensional objects, such as bars or strings, we an simply use two significant discrete points `p1 p2` and as long as $b3 = 0$ the third point is irrelevant.

3 Audio Force Models

In order to create sounds with a modal model we need good interaction force models to provide an input to the real-time algorithm described in Section 2.3. The **SonicObject** contains a single **AudioForce** object for the input excitation force, though it could easily be extended to contain many which operate on different locations. In this section we consider four types of interaction models:

- Impact forces, used for collision sounds.
- Continuous contact forces for sliding and rolling.
- Combustion engine forces.
- Live data streams.

3.1 Impact Forces

When two solid bodies collide, large forces are applied for a short period of time. The precise details of the contact force will depend on the shape of the contact areas as well as on the elastic properties of the involved materials. For example, a rubber ball colliding with a concrete floor will experience a contact force which will increase faster than linearly with the compression of the ball, because the contact area also increases during the collision. A generic model of contact forces based on the Hertz model and the radii of curvatures of the surfaces in contact was considered in (Johnson, 1985). A Hertzian model was also used to create a detailed model of the interaction forces between the mallet and the bars of a xylophone (Chaigne & Doutaut, 1997).

To create a simple model of a collision force to drive the audio synthesis, we assume that the two most important distinguishing characteristics of an impact on an object is the energy transfer in the strike and the "hardness" of the contact. A psychophysical study of perceived mallet hardness (Fried, 1990) of xylophones showed that this is indeed a very perceptible parameter of an acoustic event. The hardness translates directly in the duration of the force, and the energy transfer translates directly in the magnitude of the force profile.

We have experimented with a number of force profiles and found that the exact details of the shape are relatively unimportant. A very simple phenomenological model of a finite duration impact force can be constructed from a single period of a cosine function. The model which we implemented in a class **BangForce** approximates the contact force by a function of the form

$$F(t) = F_{max}(1 - \cos(\frac{2\pi t}{T})) \tag{7}$$

for $0 \leq t \leq T$, with T the total duration of the contact. This function has the qualitative correct form for a contact force. The force increases slowly in the beginning, representing a gradual increase in contact area, and then rises rapidly, representing the elastic compression of the materials. The sounds of soft contacts (with large T) are recognizable as such, which shows that this model can produce this perception.

In order to hit a running **SonicObject** with a **BangForce** you call `hit()` on the **BangForce** and the next time the **SonicObject** calls `getForce()`, a strike profile is returned. Typically, a duration of about $50ms$ will be perceived as a very soft "thud", and anything with a longer duration is too smeared out in time to be audible.

More complex interactions during contact may have an important effect in some cases. For example, the hammers in a piano are covered with felt, so during the contact between the hammer and the string they damp the higher modes of vibration. How this actually occurs is quite complicated (Hall, 1986, 1987a, 1987b; Stulov, 1995) and potentially important, especially for high quality musical instrument modeling. As another example, experimental data (Stoianovici & Hurmuzlu, 1996) shows that there are sequences of very fast contact separations and collisions during hard impacts. These micro collisions are caused by modal vibrations of the objects involved, and can be simulated by a short burst of impulse trains at the dominant modal frequencies (Doel et al., 2001).

The **DemoBellStrike** class contains a demonstration of the impact forces. It tolls a bell with a **BangForce** whose duration can be set with a slider. Of course, if you don't like bells, any modal object can be substituted for the bell. A second slider allows you to set the number of modes used, to get a feeling for the degradation in quality when using fewer modes. Its `main()` method contains the lines

```
sob1 = new SonicObject(new ModalModel(args[0]),
                  srate,bufferSize,bufferSizeJavaSound);
af1 = new BangForce(srate);
sob1.setAudioForce(af1);
sob1.start();
new HelperThread().start();
new DemoBellStrike (new java.awt.Frame (), true).show ();
```

which creates a **SonicObject** loaded from a modes file, assigns a **BangForce** to it and launches a control thread which hits the bell:

```
while(true) {
    sleep(t_control);
    af1.hit();
}
```

When you move the upper slider, the mallet hardness is set by the user interface thread by

```
af1.setDuration(x);
```

where `x` is set by a slider.

The same bell model is also implemented in C, without a user interface, using the PABLIO audio library for rendering.

3.2 Continuous Contact Forces

An important ingredient in synthesizing realistic scraping and rolling sounds is a surface interaction model. A lot of research has been conducted on models of contact interactions between solids (Suresh Goyal, 1994a, 1994b; Løtstedt, 1984; Keller, 1986; Baraff, 1992a, 1992b), but they usually focus on predicting forces at a coarser time scale than needed for our purposes, though not all (Ullrich & Pai, 1998). Nevertheless a rigid body simulator is able to provide information about the contact force magnitudes and the friction forces at the contact areas which may be used as inputs to a contact force model.

This model should be able to generate contact forces at the audio sampling rate for a specific type of contact given the contact force at the simulation rate (usually orders of magnitude slower than the audio rate) and the sliding speed. The roughness profile of both surfaces will determine the effective force stimulus to the object and therefore have an important effect on the sound.

A solution we found to be satisfactory is to use a looping digital sample with pitch shifting and volume control to adjust the speed and force of the contact. The physical picture behind this contact model is that the sample encodes the shape of the surface and the object scraping it is following this surface profile exactly, like the needle in a phonograph. Note that aliasing will occur for large values of the looping speed which we find to make a positive contribution to the effect in many cases, creating a richer variation with speed. With a small set of short samples representing a variety of textures a great variety of contact surface profiles can be imposed upon the vibration models. Surface profiles were created by simply scraping a real object with a contact microphone.

Another solution is to directly synthesize the contact force from a stochastic noise model, which is quite involved and will not be described here, see (Doel et al., 2001).

The Java class **LoopForce** represents a looping audio sample with controllable looping speed and volume. Its constructor reads an audio file and calls to `getForce()` return buffers obtained by looping through the sample buffer at the current speed. The method `getNextSample()` implements pitch shifting by linear sample interpolation. Experiments with a more accurate quadratic sample interpolation algorithm did not result in an audible improvement.

The demo class **DemoMouseScrape** allows you to scrape a modal model with the mouse. It samples the mouse position at a fixed sampling rate (say $50Hz$) and uses a lowpass differentiating filter to obtain the mouse velocity, which is used to set the looping speed of the excitation.

3.3 Engine Forces

Engine sounds are very difficult to achieve in computer games, as they are essentially continuous sounds which need real-time control. Racing games are very popular and a properly modeled car sound that responds in a realistic way to input parameters would greatly enhance the audio in such games.

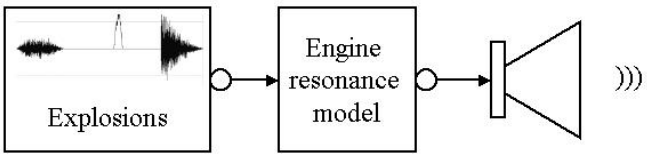

Figure 5: A hand-constructed looping wav file represents the four strokes of a simple engine model. The rest of the car including muffler is modeled as a lumped resonance model.

It is not obvious that combustion engines can be modeled with our techniques, as the sources of sound are explosions and very complicated gaseous phenomena. Rather surprisingly, we found that reasonably convincing sounding interactive models can be made by driving some resonance object (a lumped model of everything that is vibrating) with a rather simple-minded model of a combustion engine.

The first model we created is a 4-stroke engine. The driving force is obtained by constructing a looping audio sample divided into four regions which represent the 4 stages of the engine. The sample driving the resonances as depicted in Fig. 5 (`car1.wav`) contains an intake stroke, a compression stroke (silence), a combustion stroke, and an exhaust stroke. The intake stroke was modeled as white noise enveloped with a bell curve. The exhaust stroke is modeled as white noise, rapidly decaying in time, inspired by a high pressure gas mixture being released when the valve opens. The combustion stroke consists of an enveloped burst of $1/f$ noise. It was found, after trying various $1/f^{\alpha}$ noises, that this gives the most realistic sound. The reason is perhaps that the combustion takes place inside the cylinder, so the shock wave is transmitted to the mass of metal of the engine block, which acts as a low pass filter.

The sample is looped at adjustable rate, corresponding to the running speed of the engine. For added realism the engine is allowed to "misfire" once in a while, which we implement by skipping a section in the looped sample. The Java class **CombustionEngineForce** is a subclass of **LoopForce**, and has the additional method `setBeatupness(double prob)` which sets the probability per call to `getForce()` that the engine misfires.

This simple driving force model can generate a variety of engine sounds by coupling it to various vibration models. Models with relatively high frequency resonances with high damping give a "lawn mower" effect, whereas a low frequency object gives a motorcycle sound.

A slightly different sound is produced by adding a background pitched sound to the sample at all four stages. This simulates the sound of the fan, and perhaps other rotating parts. For the pitched sound, a short noisy note played on a Persian ney (a type of flute) was used, which is very satisfactory. The audio file is `car2.wav`.

Yet another engine sound was obtained by creating a four cylinder version. We assume the four cylinders fire 90^0 out of phase and simply add the samples from the one cylinder engine four times, with a relative phase shift of 90^0. If we just use this driving force as-is, the result is not very good. The reason is probably that in a real engine the four cylinders are attached to the intake manifold at different locations and therefore sound different. To incorporate this we adjust the volumes of the four one-cylinder samples individually and

when they are set all differently the resulting sound is much better. The audio file is `car3.wav`.

The demo class **DemoEngine** creates a **SonicObject** off a modes file, `car1.sy` for example, and applies a **CombustionEngineForce** to it which is loaded from an audio file. The top slider sets the speed and the bottom slider sets the "beatupness". We noticed that the result is strongly dependent on the type of audio speakers used for playback, so you may want to tune the model parameters to your audio setup.

In a real game application one would certainly spend a lot more time on the actual modeling of the car engine than we have done here. But our simple models show that a reasonable result can already be obtained using extremely simple models. To create richer engine sounds, different cylinders may be coupled to different exhaust manifolds or muffler pipes.

3.4 Live Data Streams

An interesting interface to this type of synthesis is a sensor that measures real interaction forces. This can be demonstrated with a contact microphone. When touching and scraping real objects the audio signal can be sent to a synthesis process, where this audio signal is then interpreted as a force to whatever vibration model is currently loaded. We can then scrape some interface object and transfer the measured signal to the audio synthesis to create the impression of touching a virtual object. Another type of application is to use the output of an electrical guitar as the driving force for some virtual guitar body.

The Java class **MikeAudioForce** obtains its output "force" buffer from the audio input of the sound-card:

```
protected RTAudioIn pai;
    public void getForce(double [] output, int nsamples) {
        pai.read(output,nsamples);
    }
}
```

It makes use of the utility class **RTAudioIn** which provides a convenient access to the audio input through JavaSound. The class **DemoMikeIn** loads a **ModalSonicObject** from a given modes file and attaches a **MikeAudioForce** to it. It works well when using a contact mike for scraping various surfaces. You can also plug the output of your electrical guitar in, and have a modal guitar effects box.

4 Conclusions

We have given a theoretical and practical introduction to modal synthesis, which we believe to be very useful for the real-time synthesis of a variety of sounds. After discussing the physics leading to the model, we derived an efficient algorithm, which turns out to be a familiar bank of reson filters. A Java class **ModalSonicObject** implements a modal model which can render itself. It contains an object which implements the **AudioForce** interface, which provides it with its input.

A **SonicObject** must implement its own **AudioForce** interface which allows one **SonicObject** to act as the input to another, creating in effect a filter graph.

Force models to excite the modal models were discussed and examples were given of impact forces, scraping forces, combustion engine forces, and live data streams.

The efficiency of the synthesis algorithm was measured with the provided class **DemoModalBenchmark** and we found that on a $450Mhz$ Pentium III one can synthesize about 800 modes at a sampling rate of $22050Hz$ using Java, whereas C is about 25% more efficient.

The Java code examples presented here for tutorial purposes are, with slight modifications for presentational purposes, part of JASS, which is a unit generator based audio synthesis programming environment written mainly in Java (Doel & Pai, 2001), but which also includes platform specific code for low latency audio on Windows, LINUX, and Macintosh OS X (CoreAudio). The JASS development environment is available for non-commercial use from the JASS website `www.cs.ubc.ca~/kvdoel/jass` (Doel, 2003), where you can also try out several interactive audio synthesis applets which run in all modern web browsers.

References

Baraff, D. (1992a). *Dynamic simulation of non-penetrating rigid bodies.* Unpublished doctoral dissertation, Cornell University.

Baraff, D. (1992b). Dynamic simulation of non-penetrating flexible bodies. *Computer Graphics*, *26*(2), 303–308.

Chaigne, A., & Doutaut, V. (1997). Numerical simulations of xylophones. I. Time domain modeling of the vibrating bars. *J. Acoust. Soc. Am.*, *101*(1), 539–557.

Chaudhary, A., Freed, A., Khoury, S., & Wessel, D. (1998). A 3-D Graphical User Interface for Resonance Modeling. In *Proceedings of the international computer music conference.* Ann Arbor.

Cook, P. (2003). Introduction to Physical Modeling. In K. Greenebaum (Ed.), *Audio anecdotes* (p. TODO). Natick, MA: A. K. Peters.

Cook, P. R. (1996). Physically informed sonic modeling (PhISM): Percussive synthesis. In *Proceedings of the international computer music conference* (pp. 228–231). Hong Kong.

Doel, K. v. d. (1998). *Sound synthesis for virtual reality and computer games.* Unpublished doctoral dissertation, University of British Columbia. (www.cs.ubc.ca/~kvdoel/publications/thesis.pdf)

Doel, K. v. d. (2003). *JASS Website, http://www.cs.ubc.ca/~kvdoel/jass.*

Doel, K. v. d., Kry, P. G., & Pai, D. K. (2001). FoleyAutomatic: Physically-based Sound Effects for Interactive Simulation and Animation. In *Computer graphics (acm siggraph 01 conference proceedings).* Los Angeles.

Doel, K. v. d., & Pai, D. K. (1996). Synthesis of Shape Dependent Sounds with Physical Modeling. In *Proceedings of the International Conference on Auditory Display 1996.* Palo Alto.

Doel, K. v. d., & Pai, D. K. (1998). The Sounds of Physical Shapes. *Presence*, *7*(4), 382–395.

Doel, K. v. d., & Pai, D. K. (2001). JASS: A Java Audio Synthesis System for Programmers. In *Proceedings of the International Conference on Auditory Display 2001.* Helsinki, Finland.

Doel, K. v. d., Pai, D. K., Adam, T., Kortchmar, L., & Pichora-Fuller, K. (2002). Measurements of Perceptual Quality of Contact Sound Models. In *Proceedings of the International Conference on Auditory Display 2002.* Kyoto, Japan.

Fried, D. J. (1990). Auditory correlates of perceived mallet hardness for a set of recorded percussive sound events. *J. Acoust. Soc. Am.*, *87*(1), 311–321.

Gaver, W. W. (1993). Synthesizing auditory icons. In *Proceedings of the acm interchi 1993* (pp. 228–235).

Hall, D. E. (1986). Piano string excitation in the case of small hammer mass. *J. Acoust. Soc. Am.*, *9*(1), 141–147.

Hall, D. E. (1987a). Piano string excitation II: General solution for a hard narrow hammer. *J. Acoust. Soc. Am.*, *81*(2), 535–545.

Hall, D. E. (1987b). Piano string excitation III: General solution for a soft narrow hammer. *J. Acoust. Soc. Am.*, *81*(2), 547–555.

Johnson, K. L. (1985). *Contact mechanics.* Cambridge: Cambridge University Press.

Keller, J. B. (1986). Impact with friction. *Journal of Applied Mechanics*, *53*(1), 1–4.

Klatzky, R. L., Pai, D. K., & Krotkov, E. P. (2000). Perception of material from contact sounds. *Presence*, *9*(4), 399–410.

Krotkov, E., & Klatzky, R. (1995). Robotic Perception of Material: Experiments with Shape-Invariant Acoustic Measures of Material Type. In *Preprints of the fourth international symposium on experimental robotics, iser '95.* Stanford, California.

Løtstedt, P. (1984). Numerical simulation of time-dependent contact and friction problems in rigid body mechanics. *SIAM J. Sci. Stat. Comput.*, *5*(2), 370–393.

Mathews, M. V. (1969). *The technology of computer music.* Cambridge: MIT Press.

Morrison, J. D., & Adrien, J.-M. (1993). Mosaic: A framework for modal synthesis. *Computer Music Journal*, *17*(1).

O'Brien, J. F., Chen, C., & Gatchalian, C. M. (2002). Synthesizing Sounds from Rigid-Body Simulations. In *Siggraph 02.*

O'Brien, J. F., Cook, P. R., & Essl, G. (2001). Synthesizing Sounds from Physically Based Motion. In *Siggraph 01.* Los Angeles.

Pai, D. K., Doel, K. v. d., James, D. L., Lang, J., Lloyd, J. E., Richmond, J. L., & Yau, S. H. (2001). Scanning physical interaction behavior of 3D objects. In *Computer Graphics (ACM SIGGRAPH 01 Conference Proceedings).* Los Angeles.

Serafin, S. (2003). Physical Synthesis of Bowed String Instruments. In K. Greenebaum (Ed.), *Audio anecdotes* (p. TODO). Natick, MA: A. K. Peters.

Smith, J. O. (1992). Physical modeling using digital waveguides. *Computer Music Journal*, *16*(4), 75–87.

Steiglitz, K. (1996). *A digital signal processing primer with applications to digital audio and computer music.* New York: Addison-Wesley.

Stoianovici, D., & Hurmuzlu, Y. (1996). A critical study of the applicability of rigid-body collision theory. *ASME Journal of Applied Mechanics*, *63*, 307-316.

Stulov, A. (1995). Hysteretic model of the grand piano hammer felt. *J. Acoust. Soc. Am.*, *97*(4), 2577–2585.

Suresh Goyal, F. W. S., Elliot N. Pinson. (1994a). Simulation of Dynamics of Interacting Rigid Bodies Including Friction I: General Problem and Contact Model. *Engineering with Computers*, *10*, 162–174.

Suresh Goyal, F. W. S., Elliot N. Pinson. (1994b). Simulation of Dynamics of Interacting Rigid Bodies Including Friction II: Software System Design and Implementation. *Engineering with Computers*, *10*, 175–195.

Ullrich, C., & Pai, D. K. (1998). Contact response maps for real time dynamic simulation. In *Proceedings of the ieee international conference on robotics and automation* (pp. 1019 – 1025). Leuven.

Wawrzynek, J. (1989). VLSI models for real-time music synthesis. In M. Mathews & J. Pierce (Eds.), *Current directions in computer music research.* MIT Press.

Wildes, R. P., & Richards, W. A. (1988). Recovering material properties from sound. In W. Richards (Ed.), *Natural computation.* Cambridge, Massachusetts: The MIT Press.