Physically Based Sound Synthesis for Interactive Applications

by

Benen Cahill, B.A.

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science

University of Dublin, Trinity College

September 2009

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Benen Cahill

September 18, 2009

In memory of Mai, Teresa and Keith.

Acknowledgments

Thanks to John for all his patience, and to Rosemarie, Henry and Finbarr for all their support. And to my family, for being with me throughout.

Benen Cahill

University of Dublin, Trinity College
September 2009

Physically Based Sound Synthesis for Interactive Applications

Benen Cahill
University of Dublin, Trinity College, 2009

Supervisor: John Dingliana

Traditional audio techniques for interactive applications typically involve the capture and playback of numerous recordings. Such approaches are expensive, time consuming and unfeasible for adequately accompanying the ever expanding visuals of an interactive system. Physically based sound synthesis is an alternative to traditional techniques that has the potential for additional realism and immersion in audio. It describes a framework for the synthesis of realistic impact sounds in real time, based on an object's rigid body motions. It is achieved by modelling an object's deformations when subject to external forces, and subsequently solving for the displacements to produce a sound wave. However, it is a decidedly expensive task in terms of computation and implementation.

The contribution of this work is to find new methods to simplify the process and ultimately render an implementation more viable. Two approaches are taken. First, existing methods for the creation of deformation models are explored in detail. A new approach is defined through the combination of existing techniques. Secondly, parallelisation of the synthesis is explored, and a robust, coarse-grained approach is presented, with the capacity for significant performance improvements. Finally, a full synthesis pipeline and interactive application is developed to evaluate the results.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Sound and vision are inseparable. Human experience of the world would not be complete without one or the other. For example, consider the terror that can be caused by thunder and lightning. The lightning itself, while beautiful in many forms, would not be as awe-inspiring but for the thunderous roar that accompanies it. Our existence is defined just as much as the sounds we hear, as the sights we see.

Despite this, in the world of interactive applications, audio tends to be a much less understood component. Indeed, most audio techniques in interactive applications borrow heavily from the film industry. The result is a strange mix and match between artistry, engineering and computer science. Recordings of real world sounds form the basis and can take hundreds if not thousands of man-hours to capture. The next requirement is to create sophisticated algorithms to allow us to mix and match these recordings in order to create the desired audio effects. And lastly, is the issue of playback, where memory management becomes the primary concern. The end product is frequently a mix of looped sounds, unsubtle triggers and a bizarre discontinuity between the realism of the rendered visuals and the recorded audio. It just does not fit.

Consider, for a moment, the medium of film. In film it is not unusual to see the storyline advanced through audio alone. For example, numerous thrillers feature the same scene with the heroine walking through a dark room, stalked by an enemy who is only given away by the ambient noises she creates. Every movement made, every step taken, risks drawing the attention to the heroine. Thus, the audio is pushed to the forefront of the viewer's attention, and every single background noise sends shivers down the spine. In contrast, for games the audio typically serves little more than to complement the environment and establish the mood of the player. The player relies not on the audio to form the game-play, only where to point their visual attention. This is because it is not feasible to create such a sophisticated soundtrack through recordings for an interactive medium with near infinite possible outcomes for the game-play. Yet, we still rely on audio recordings, much to the detriment of the perceptual experience. One would never create the visuals of an interactive application through videos and pictures.

Despite the current limitations in the techniques, many within the industry recognise the importance audio production will play in the future of the interactive applications [2]. Indeed there are already a waft of new experimental techniques for producing realistic game audio in the literature. One of these methods is termed Physically Based Sound Synthesis, and forms the scope of this work.

1.2 Physically Based Sound Synthesis

Physically Based Sound Synthesis is the process of synthesising impact sounds through rigid body motions. Its foundations lie deep in the theory of vibration. Briefly, when an object is struck, the forces exerted on it can be decoupled into two main components. The first is the 'dynamic force' which dictates how the object will move in response (*i.e.* does it fall over, roll along the ground, get hurled in the air?). These 'dynamic forces' are typically modelled in modern interactive applications through the use of 'rigid body simulators' (or 'physics engines'). The second component is the elastic response (for example, stretching, compression or vibration). It is through these elastic forces that sound is produced in real time. Typically when an object is struck, it will vibrate. Frequently these deformations are too small to see with the naked eye. However, they cause displacements in the medium around it and pressure waves are radiated from the object. These pressure waves are what humans perceive as sound.

Physically based sound synthesis is the process of modelling these elastic displacements in order to synthesize sound. First, a model is created that describes how the objects vibrate. Then, forces are captured from the rigid body simulator and used to excite these deformations. Finally, we solve for the displacements to produce a sound wave and subsequently render the audio response of the object. The model will be explained in detail in the following chapters. For now, it is sufficient to note that physically based sound synthesis is the field of investigation for this work.

1.3 Objectives

Previous work showed how inaccessible physically based sound can be [3], despite a recent surge of developments in the area. There are a variety of different reasons for this:

- Physically based sound represents a completely different paradigm to standard game audio programming. Traditional methods focus on recording sounds and mixing their playback at run time to create the different sound effects. Physically based sound, however, requires extensive knowledge of vibrational theory and digital signal processing, techniques not common in traditional solutions.
- Previous work has, to a large degree, used relatively expensive proprietary software and hardware throughout their implementations, for the creation of the models, and in some cases, the rigid body simulator. For example, van den Doel uses a specific device that was constructed to model an object's impulse response. This has largely prevented the distribution of any tools or libraries, which, through their absense, have created a barrier to entry in the area. Indeed, there are only two libraries that we know of for the purposes of synthesis, JASS [30] and STK [36], and neither is really appropriate for interactive applications.
- There is a distinct lack of comprehensive documentation in the area. To this author's knowledge, there are no textbooks or indeed meta-analyses of the work that has been done, making it difficult in cases to locate and access the required information. This leaves users working from a limited set of resources, and compounds the barriers to the potential developer.

Thus, it is the intention for this work to focus on finding methods of simplifying physically based sound synthesis, and in particular to focus on methods that would be suitable for developers outside academia who may not have the resources at their disposal to implement previous work. As this author demonstrated in previous work [3], the greatest difficulty towards an implementation is presented through the generation of the models. On the back of this, the generation of modal models will be explored in detail.

Specifically, this study proposes to explore the possibility of combining simple mass springs models with data from recordings. The author's previous work has shown that the creation of modal models through mass springs do respond in a physically correct manner to input [12], however, there is a significant loss in quality in the timbre of an object. Thus, possible improvements to the existing algorithms for recorded data will be explored, to see if they can be combined in some manner with the physically based models. If successful, it would represent a method cheap in both resources and man-hours to create realistic and immersive audio.

Secondly, possible methods for improving the efficiency of synthesis will be explored. It has been shown in previous work [23] that physically sound synthesis is a highly parallelisable task and suitable for implementation on the graphics processing unit (GPU). However, due to the surge in general purpose computation on the GPU (GPGPU) the graphics card is already the major bottleneck in most systems. Thus, alternative parallelisation approaches on the central processing unit (CPU) will be explored.

Finally, it has been noted from reading the literature that the current state of the art is highly fragmented and, in many respects, inaccessible to the lay user. One of the possible reasons for this is the lack of any standardised textbook or general platform for the field. Thus it is a secondary intention for this work to explore the literature in detail and provide something of an evaluation. This will be performed through implementing a full synthesis pipeline, in order to identify the current strengths and weaknesses of the state of the art.

1.4 Outline of the Document

The rest of the document is structured as follows:

Chapter 2 gives a brief overview of the modal synthesis model. It serves as an introduction to the main concepts used throughout the rest of the document. It describes the foundations of the model and introduces the reader to the modal synthesis pipeline.

Chapter 3 discusses the main methods for generating the modal models. It describes the three main techniques used extensively throughout the literature: that is, analytical models, numerical models and models generated from recorded data. In this chapter we also briefly discuss some of the models that have been proposed for liquid sounds, as well as issues of propagation and emission.

Chapter 4 explains how the input forces for the modal model are obtained. It discusses how to resolve the synthesis pipeline with the rigid body simulation and the rendering pipeline, and presents the various methods in the literature for modelling different types of impact sounds.

Chapter 5 highlights the main concerns of synthesis and the rendering of the audio. It presents a number of different models from the literature for performing efficient synthesis, and describes additional methods for reducing the overall computational burden.

Chapter 6 is where the main contribution of this work begins. The basic design of a modal synthesis pipeline is discussed, and how one can derive efficient techniques.

Chapter 7 discusses the implementation, and how the pipeline was assembled.

Chapter 8 gives a detailed breakdown of the results obtained from the implementation. The overall perceptual quality of the implementation is evaluated, as well as the performance and weaknesses of the implementation.

Chapter 9 comprises the conclusion. Our main achievements and findings from the work are outlined. Possibilities for future directions of research are also listed.

It should be noted that the literature review is conducted in Chapters 3 through 5. This approach was taken so as to clearly identify the primary concerns of the model, of which familiarisation is necessary in order to understand the aspirations for this work.

For those readers not familiar with the field of Digital Signal Processing, it is suggested to read Appendix A before proceeding, as it explains a number of concepts used extensively throughout the document.

Another important note involves the mathematical notation used throughout this text. Where possible, it was attempted to standardise the notations used. However, this proved difficult as the symbols meaning tends to change in relation to the context. Thus, it is not unusual to see the complex frequency being referred to as ω or Ω.

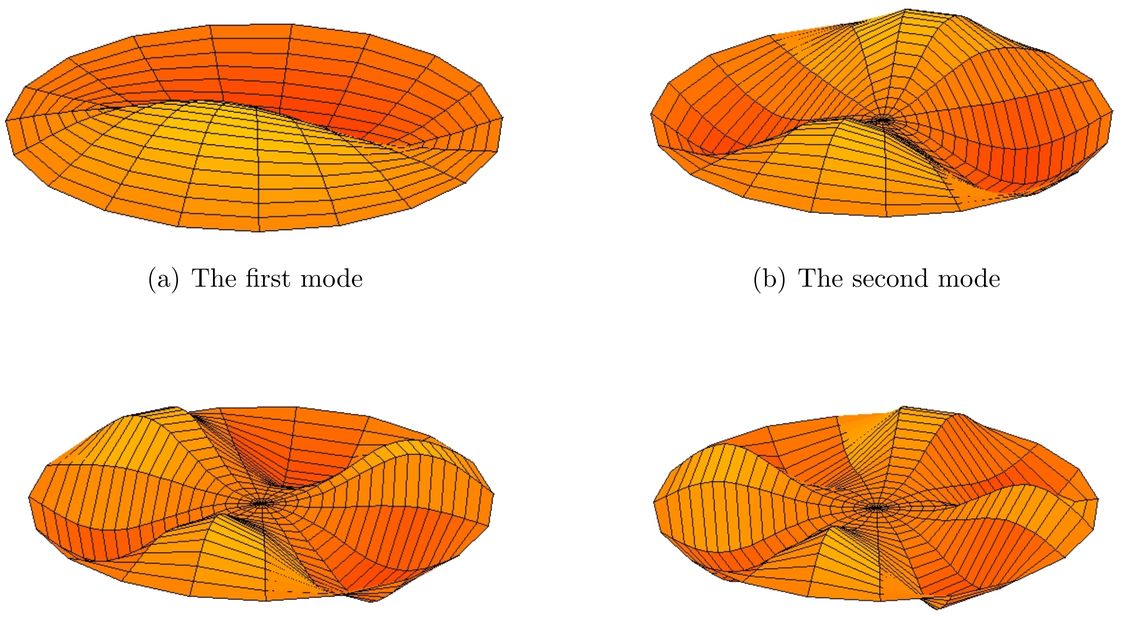
Chapter 2

Background

2.1 A Note on Sound

Sound is simply pressure waves in the air. All sounds that we hear are typically the result of vibrations or displacements in an object's surface that excites the air around it and produces the pressure waves that humans perceive as sound. This is because almost all materials have some elastic properties, so that when a force is exerted on them, the elastic forces within that object cause it to vibrate until it comes to rest. Typically these deformations are so small that we cannot see them, however, we need only look at the vibration of a plucked guitar string or a drum when struck to see this process taking place.

How an object vibrates is extensively researched in the area of vibrational theory [18]. What has been discovered is that there are a variety of different patterns of vibration in an object, inherent to its geometry and material properties. These vibrational patterns are called the 'modes of vibration' or natural frequencies of the object in question. For example, see Figure 2.1 which shows the first four vibrational modes of a circular drum skin. Analytical solutions have shown that the number of vibrational modes an object contains is infinite [19]. However, through observation we find that there are only a few characteristic frequencies that determine an object's overall sound. Typically, on impact, a very high number of vibrational modes are excited. Yet most of these modes decay almost instantaneously, particularly those with very high frequencies, leaving only a few modes that vibrate long enough to be heard. For example, for a musical instrument, or an object that exhibits a definite pitch (like a metal cylinder), the pitch is due to a single mode decaying much slower than all the others. The frequency of this mode is what determines the pitch.



(a) The first mode

(b) The second mode

(c) The third mode

(d) The fourth mode

Figure 2.1: The vibrational modes of a circular membrane

Physically based sound synthesis is the process of modelling these vibrations, or deformations and then directly synthesising them to produce sound. This is achievable because the vibrational modes of an object are shown to exhibit strong sinusoidal properties. Each mode has a fixed frequency and exponential decay, which is determined by the internal friction forces of the material in question. As such, almost all forms of physically based sound synthesis model these vibrational modes a set of linear harmonic oscillators. The synthesis then is simply a process of finding the response of each mode to a given impact force.

2.2 A Harmonic Oscillator

Before the reader is introduced to the modal synthesis model, the properties of a linear harmonic oscillator shall be discussed. A harmonic oscillator is any object that when displaced, experiences a restoring force until it reaches equilibrium. Thus, it will oscillate about its rest position, until such point that its energy has been dispelled, for example, through heat, sound or friction. The restoring force is governed by Hooke's Law:

$$F = -kx - c\dot{x} \tag{2.1}$$

where the force, F is determined by the elastic force, k, and the displacement, x. In this case we have introduced a dampening value, c. By combining this with Newton's Second Law of Motion ($F = ma$) we arrive at:

$$m\ddot{x} + c\dot{x} + kx = 0 \tag{2.2}$$

This second order, ordinary differential equation (ODE), gives the rate of change of the oscillator's displacement. The solution to this equation is given by:

$$x_t = x_0 e^{(ct/2m)} \cos\left(t\sqrt{(k/m - (c/2m)^2)}\right) \tag{2.3}$$

Thus, the solution to any linear harmonic is a sinusoid. And as most sound waves can be modelled as sinusoids, we have arrived at a powerful method for synthesis.

2.3 Modal Synthesis

The modal synthesis model is presented as follows:

$$y_t = \sum_{n=1}^{N} a_{nk} e^{-d_n t} sin(2\pi f_n t) \tag{2.4}$$

This relation bears a close resemblance to that of the linear harmonic oscillator above. Indeed, the precise methods used to identify an object's modes are performed in a manner similar to the steps above.

The displacements of the objects N modes, y_t, produce pressure waves in the air that we perceive as sound. The displacements are determined by the natural frequency (f in Hertz, or ω in radians), the dampening parameter d, and the amplitude or gain for each mode, a_{nk} (in other texts a_{nk} is represented by c_n). The frequency and dampening of a mode remain constant, however, the value of the mode's amplitude depends on the impact force and the geometric location of where the force is exerted on the object. Under the assumption of linearity, the precise shape of the modal displacement does not matter, however, it should be noted that this only results in an approximation of each mode's displacement. In reality, non linearity is frequently observed. However, by and large, the assumption of linearity is perfectly acceptable for synthesising sound.

The amount of modes any particular object contains is infinite, and stretches to infinitely high frequencies. However, in reality only an impulsive force of infinitely short duration will excite all the modes. And given the limits of human perception, we can eliminate many of these modes, as we need only concern ourselves with modes that oscillate with a frequency in the audible range (that is, 20Hz to 22000Hz).

In order to construct the audio signal of an object's impact, it is then a simple case of evaluating the above expression at each point in time, t. This is referred to in the literature as 'additive synthesis', whereby an audio signal is constructed by summing a set of sinusoids. However, it is an expensive task, due to Nyquist's law [8, 15]. This states that in order to reconstruct a signal of a frequency, f, we must sample the signal at a rate of at least $2f$, as failure to do so will result in the inability to capture the crest and trough of each wave, which renders the signal inaudible. However, given that the upper limit of human audio perception is 22000Hz, it means in order to recapture most objects adequately, we must evaluate Equation 2.4 over 44000 times per second. Thus, modal synthesis can easily become a computationally expensive task.

2.4 Modal Synthesis Pipeline

The entire modal synthesis pipeline can be described briefly as follows:

First, we identify an object's deformation modes. We then assemble this into a modal model of the form $M = \mathbf{f}, \mathbf{d}, A$. The precise form depends greatly on the method of identification used, however, it generally consists of column vectors $\mathbf{f}$ and $\mathbf{d}$ that describe the frequency and dampening parameters. A is a matrix that is used

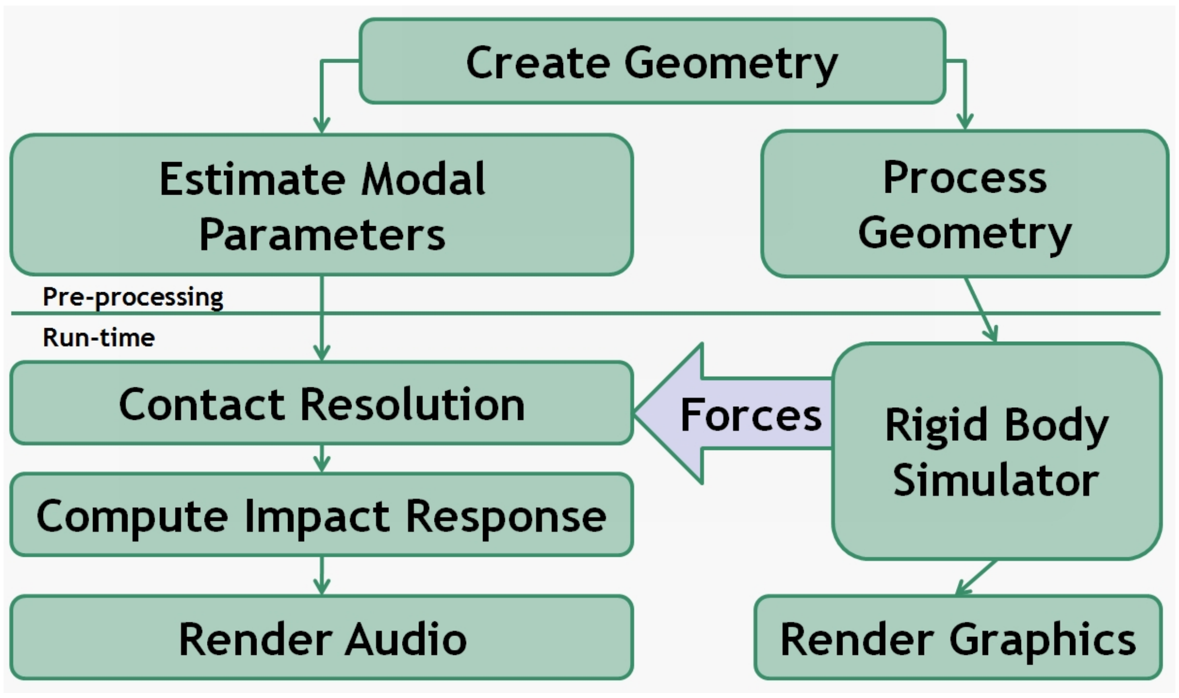


Figure 2.2: A high level diagram of a full synthesis pipeline

to specify the gains for each mode depending on the contact location, k. The precise methods are discussed in Chapter 3.

The next step is to introduce some adequate contact modelling. This refers to the process of extracting the forces from the rigid body dynamics and converting them into a form suitable for audio synthesis. This is a necessary step, as rigid body simulators run at much lower frequency than the audio processing (60Hz, as opposed to 44100Hz). The precise method again will depend on the nature of the modal model, and the main methods for doing so are discussed in Chapter 4.

Finally, we solve the modal displacements for each audio sample, and render the audio. This is an expensive process, and to do it efficiently we need to use additional techniques, as opposed to solving the relation above, typically in the form of recursive filters. These steps, and a number of other speed-ups for the synthesis, are discussed in Chapter 5.

Overall, the process is analogous to the rendering pipeline, and is illustrated in Figure 2.2.

It may be noted that there has been no mention of propagation. Propagation is the term used to describe how sound waves are scattered throughout its environment, and how they can be disturbed and manipulated by the objects around them. For example, the propagation of sound in a very large space will typically cause an echo, as it reflects off objects and bounces back to the listener at different rates, depending on the route that needs to be travelled. Typically, however, most implementations thus far have treated the output of modal synthesis as a point source. Indeed, this makes for much more simple integration of modal synthesis with traditional audio pipelines. However, there have been some models created in the literature, and these are briefly discussed in Chapter 3, although they were not used in this implementation.

Chapter 3

Generating Modal Models

At the simplest level, generating modal models for real-time applications involves creating a bank of sinusoids, that when synthesised are added together to render the characteristic sound of a rigid body when struck. There are quite a number of options for the creation of models.

Van den Doel et al. showed how it is possible to solve the equation of motion for simple shapes by hand, and thus construct the frequencies and dampenings [22]. Using these simple solutions for an object's faces (that is, the faces defined in the three dimensional mesh), it is then possible to model a variety of different modal models.

An alternative is to extract the modes using recordings. By performing a Fourier transform on a signal, it is possible to identify the contributing sinusoids, and thus generate a modal bank.

For more complex geometry, Finite Element Methods (FEM) tend to be the preferred methods, and indeed tend to be far more robust than other methods. The following sections go into more details about how one can generate modal models, and thus make an informed decision for a particular application.

3.1 Analytical Methods

Doel and Pai demonstrated how to generate the parameters needed for modal synthesis through analytical solutions of simple shapes[21][22][19]. This is done by assembling the shape's equation of motion, solving the system and then determining the amplitudes of vibration for each mode depending on impact location.

For illustrative purposes, they describe how to create sounds from a rectangular membrane. The solution to this is well documented in literature on vibrational theory. The vibration response of the object can be described as $\mu(x,t)$. It is assumed that the vibrational response takes the form of a wave equation:

$$(A - \frac{1}{c^2}\frac{d^2}{dt^2})\mu(x,t) = F(x,t) \tag{3.1}$$

In this case A is the differential operator that corresponds to the shape in question. For example, for the rectangular membrane, A is of the form:

$$A = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \tag{3.2}$$

To solve the equation, we must give it some initial conditions:

$$\mu(x,0) = y_0(x) \tag{3.3}$$
$$\frac{\partial \mu(x,0)}{\partial t} = v_0(x) \tag{3.4}$$

In this case $v_0(x)$ is the initial velocity, and corresponds to a square membrane that is attached to a fixed frame, which means there will be no vibration at the edges of the membrane.

The solution to the equation is given in the form:

$$\mu(x,t) = \sum_{n=1}^{\infty}(a_n sin(\omega_n ct) + b_n cos(\omega_n ct)\Psi_n(x)) \tag{3.5}$$

where a_n and b_n are determined by the initial conditions placed on the system, $\omega_n$corresponds to the eigenvalues of the system and $\psi_n(x)$ are the eigenfunctions.

For the rectangular membrane, the eigenfunctions and eigenvalues are labelled n_x and n_y and can be found through:

$$\Psi_{n_x n_y} = sin(\pi n_x x/L_x)sin(\pi n_y y/L_y$$
$$\omega_{n_x n_y} = \pi\sqrt{(\frac{n_x}{L_x})^2 + (\frac{n_y}{L_y})^2}$$

The eigenfunctions can then be normalised so that the norm of both (α) is independent of n. This means we can define a_n and b_n as:

$$a_n = \int_S \frac{v_0(x)\Psi_n(x)}{c\alpha_n\omega_n} d^k(x) \tag{3.6}$$

$$b_n = \int_S \frac{y_0(x)\Psi_n(x)}{c\alpha_n} d^k(x) \tag{3.7}$$

And the average energy of vibration (assuming mass is uniformly distributed) is given by:

$$E = constant \times \sum_{n=1}^{\infty} \alpha_n \omega_n^2 (a_n^2 + b_n^2) \tag{3.8}$$

From the above, we have now arrived at the fundamental frequencies of the modes of vibration of the membrane. However, in order to produce sound, we need both the amplitudes and model of how the vibrations are converted to pressure waves in order to produce sound.

For a given impact point p on the surface of the membrane S, we need to compute the corresponding amplitudes. To do this we assume initial conditions of:

$$y_0(x) = 0 \tag{3.9}$$

$$v_0(x) = \delta(x - p) \tag{3.10}$$

where δ is the Dirac-delta function. This will lead to an infinite sum, however, by ignoring frequencies outside the audible spectrum we can create a finite range. The amplitudes can then be obtained as a function of the impact location:

$$a_n = \frac{\Psi_n(p)}{c\alpha_n\omega_n} \tag{3.11}$$

$$b_n = 0 \tag{3.12}$$

Then the amplitudes are scaled according to the energy of the impact.

We also need to make some assumption about how the vibration of each mode decays. A simple method is to assume each mode decays exponentially according to some internal friction parameter (ϕ) of the material in question. Thus the decay rate for each mode can be computed as:

$$\tau_i = \frac{1}{\pi f_i \tan \phi} \tag{3.13}$$

Through the above methods, we can create a sound map of the object in question, with frequencies, dampenings and amplitudes for each mode.

3.2 Numerical Methods

Numerical methods can also be used to solve for the vibrational modes of a system, and indeed in most cases are more robust and generalisable than analytical methods [10][12][11]. Indeed, the process can be described as a discretisation of the system, whereas the analytical methods are a continuous solution, so in essence we are still simply solving an object's equation of motion[10] to model its deformations under force.

Finite Element Methods (FEM) are largely the preferred choice. A simplistic definition describes it as the process of dividing a structure into elements (or pieces) [5]. This allows us to model the physical behaviour (in our case, vibration) of each element in a simplistic manner, and then reconnecting the elements at some arbitrarily defined 'nodes'. However, its applications are far more broad than the analysis of structures, and it can be used to model everything from stress analysis, sound propagation, heat propagation to the weather. Indeed, a more general definition describes them simply as a method for solving, or decoupling partial differential equations (PDE) in the domain of some spatial field.

The family of methods we are interested in, are dynamic methods, or more specifically, the family of methods labelled modal analysis. This is the process of determining the dynamic characteristics in terms of its natural frequencies and modal shapes. It is similar to the analytical methods above in that what we are interested in is modelling the vibrational response of an object. It is based on the premise that the vibrational response of an object can be described in terms of a linear combination of the simple harmonic motions of its elements.

O'Brien et al. were the first to apply FEM to the domain of physically based sound [10]. Initially, they constructed a model that allowed them to generate sound waves directly from an FEM model of the object in question. However, while arguably the most physically correct solution, the system was far too slow for real-time applications. Instead, they were able to conduct the modal analysis as an off-line pre-process, to solve for the system's fundamental frequencies (i.e. modes). These modes would then be mixed at run time (similar to van den Doel's method). The pre-process allowed them to separate the deformation model of the modes with the idealised rigid body motions of the object. This meant the system could be used with any off-the-shelf rigid body simulator.

Broadly speaking, the offline process has two stages: creating the deformation model given some geometrical representation of a physical object, and identifying the system's modes.

Deformation Model

In order to model the deformations, we need to assemble the object's equation of motion. The equation of motion of any discretised system can be given as follows:

$$K(d) + C(d, \dot{d}) + M(\ddot{d}) = f \tag{3.14}$$

In this instance, M is the mass matrix, which represents the masses associated with each element, K is the elastic force matrix and C is the internal damping matrix, which represents the forces due to node displacements. In this case, f is the external force applied to the system and d represents the vector of displacement. In order to solve for the modes of vibration of the object we will need to decouple the system into a set of linear ODEs. However, how this can be done depends on the properties of the force matrices of the system. Ultimately, they need to be real and symmetric.

3.2.1 Tetrahedral Methods

There are a variety of different methods available to discretise the system, both from surface models such as a mass-springs system to volumetric approaches (both of which shall be discussed later). The method selected by O'Brien et al.[7] uses a tetrahedral finite element method, and tetrahedral methods have subsequently become the most widely used in the area of physically based sound synthesis for rigid bodies (it is also used in [7] [1]).

To compute the stiffness and mass matrices of the system, they first compute the stiffness and mass matrices for each element. These are found using the non-linear node forces described below for a single element:

$$f_{[i]a} = \frac{-vol}{2} \sum_{j=1}^{4} p_{[j]a} \sum_{k=1}^{3} \sum_{l=1}^{3} \beta_{ji}\beta_{ik}\sigma_{kl} \tag{3.15}$$

The element's stiffness matrix (k) is then assembled by evaluating the partials of f at their rest position:

$$\begin{aligned} k_{[ij]ab} &= \frac{\partial f_{[i]a}}{\partial p_{[i]b}} \\ &= \frac{-vol}{2}(\lambda\beta_{ia}\beta_{jb} + \mu\beta_{ib}\beta_{ja} + \mu\sum_{k=1}^{3}\beta_{ik}\beta_{jk}\delta_{ab} \end{aligned} \tag{3.16}$$

The mass matrix (m) is then computed from the partials of kinetic energy with respect to the node velocities, as below:

$$\begin{aligned} m_{[ij]ab} &= \frac{\partial k^2}{\partial\dot{\rho}_{ia}\partial\dot{\rho}_{jb}} \\ &= \frac{\rho Vol}{20}(1+\delta_{ij})\delta_{ab} \end{aligned} \tag{3.17}$$

The element matrices are 12×12. These are then assembled into the corresponding global mass and stiffness matrices (M and K respectively). This is done by simply accumulating the entries in the element matrices into corresponding entries in the global matrices. For a rigid body in three dimensional space the global matrices will be of size $3N \times 3N$ (where N is the number of nodes in the tetrahedral mesh). The results are large sparse matrices. To save computation time, it is also possible to accumulate the sum of the rows in the global mass matrix along the diagonal, and then discard the original entries.

It is also possible to eliminate the need for a damping matrix through the use of Rayleigh Damping. Rayleigh damping is a non physically based method to express the global damping matrix in terms of some contribution of both the stiffness and mass matrices, as shown below:

$$C = \alpha K + \beta M \tag{3.18}$$

The choice of α and β are somewhat arbitrary, however, in general it is found to be a useful approximation and the result greatly simplifies the efforts needed to decouple the system.

Identifying the Modes

Once the global forces matrices have been assembled, the system can be expressed in the following form:

$$K(d + \alpha\dot{d}) + M(\beta\dot{d} + \ddot{d}) = f \tag{3.19}$$

This makes the assumption that the deformations are linear and small, which is sufficient for the purposes of sound synthesis as the deformations that produce sound are very small. The next step is to diagonalise the equation. This allows us to decouple the system into a set of linear ODEs which we can then solve to find the objects deformation modes. Again, how the system is solved depends greatly on how the matrices are assembled. If M is diagonal, it is possible to diagonalise K to finds its eigenvectors and eigenvalues (as we shall see later). If this assumption cannot be made however, there are other techniques available. O'Brien et al. selected Cholesky factorisation of M so that it can be expressed in terms of the product of a triangular matrix and its transpose ($M = LL^T$). This allows for the introduction of a new variable, $y = L^T d$. Rewriting in terms of y and multiplying by L^{-1} yields:

$$L^{-1}KL^{-T}(y + \alpha\dot{y}) + \beta(\dot{y} + \ddot{y}) = L^{-1}f \tag{3.20}$$

This then allows for $L^{-1}KL^{-T}$ to be decomposed to solve for its eigenvalues and eigenvectors ($V\Lambda V^T$). By introducing a new variable, $z = V^T y$ and multiplying by V^T, the equation can be rearranged to:

$$\Lambda(z + \alpha\dot{z}) + (\beta\dot{z} + \ddot{z}) = V^T L^{-1} f \tag{3.21}$$

and then again to:

$$\Lambda z + (\alpha\Lambda + \beta I)\dot{z} + \ddot{z} = g \tag{3.22}$$

where:

$$g = V^T L^{-1} f \tag{3.23}$$

Thus, the system has now been diagonalised into a set of ODEs, each of which takes the form of a harmonic oscillator.

$$\lambda_i z_i + (\alpha\lambda_i + \beta)\dot{z}_i + \ddot{z}_i = g_i \tag{3.24}$$

These equations represent the i'th mode of the system, where λ_i is the i'th entry along the diagonal Λ. The solutions of the equation are given by:

$$\begin{aligned} z_i t &= c_1 e^{\omega_i^+ t} + c_2 e^{\omega_i^- t} \\ \omega_i^{\pm} &= \frac{-(\alpha\lambda_i + \beta) \pm \sqrt{(\alpha\lambda_i + \beta)^2 - 4\lambda_i}}{2} \end{aligned} \tag{3.25}$$

where ω represents the complex frequency of the mode, from which we can derive the decay rate ($Re(\omega)$) and the angular frequency ($|Im(\omega)|$) needed for sound synthesis. c_1 represents the amplitude (or gain) of the mode. For a given impulse, c is:

$$c_1 = \frac{2\Delta t g_i}{\omega_i^+ - \omega_i^-} \tag{3.26}$$

$$c_2 = \frac{2\Delta t g_i}{\omega_i^- - \omega_i^+} \tag{3.27}$$

This means our synthesis equation reduces to the following:

$$z_i = \frac{2\Delta t g_i}{|Im(\omega_i)|} e^{tRe(\omega_i)} \sin(t|Im(\omega_i)|) \tag{3.28}$$

3.2.2 Mass Spring Methods

A simplification of O'Brien's method was proposed by Raghuvanshi et al. [12]. They use a mass-spring model derived from an object's surface geometry to create the modal models. It makes no assumption about the mesh connectivity, which allows us to use a wide variety of 3D models common to interactive applications.

The mass-spring system is created from the geometry, where each vertex is replaced by a particle and each edge is replaced by a damped spring. The masses for each particle

(m_i) and the spring constants for each spring (k) are formulated taking into account Young's Modulus of Elasticity, (Y), the density of the material the object consists of (ρ) and the thickness of the object's surface (t). They are given by the following:

$$\begin{aligned} k &= Yt \\ m_i &= \rho t a_i \end{aligned} \tag{3.29}$$

where a_i relates to the area that corresponds to a particle. It is found by dividing the area of each face by the number of its vertices, then summing the contributions for each face the particle is identified with.

The system is then assembled from each particle's equation of motion [6]. To illustrate how this is formulised we will give a simple example to solve for movement along the x-axis. Given a system with 2 particles (m_1 and m_2) and 3 springs (whose constants are denoted by k_i) the equations of motion for each particle are given by:

$$\begin{aligned} m_1\ddot{x_1} + (k_1 + k_2)x_1 - k_2x_2 &= 0 \\ m_2\ddot{x_2} - k_2x_1 + (k_2 + k_3)x_2 &= 0 \end{aligned} \tag{3.30}$$

In order to formulate the system's equation, we assemble the above equations into the mass and force matrices as below:

$$\begin{vmatrix} m_1 & 0 \\ 0 & m_2 \end{vmatrix} \begin{matrix} \ddot{x_1} \\ \ddot{x_2} \end{matrix} + \begin{vmatrix} k_1 + k_2 & -k_2 \\ -k_2 & k_2 + k_3 \end{vmatrix} \begin{matrix} x_1 \\ x_2 \end{matrix} = \begin{Bmatrix} 0 \\ 0 \end{Bmatrix} \tag{3.31}$$

Thus we can describe the system as:

$$M\ddot{x} + Kx = 0 \tag{3.32}$$

M is the diagonal matrix which corresponds to the particle masses, K is the elastic force matrix where entries correspond to springs. However, we can see from the above that this is an undamped system. Like with O'Brien's method, dampening is introduced in the form of Rayleigh damping. Thus the linear system takes the following form:

$$M\ddot{r} + (\gamma M + \eta K)\dot{r} + Kr = f \tag{3.33}$$

where γ is the fluid and η the viscoelastic damping constants. For our system of N particles in three dimensional space, the size of the mass and force matrices are extended to allow for 3 degrees of freedom for each particle, so the final size of the matrices will be $3N \times 3N$.

This system is very similar to O'Brien's, and the solution to the system is found by diagonalising K to find its corresponding eigenvalues and eigenvectors. This results in the decoupling of the system into a series of linear ODEs. The solutions for each mode are then given by:

$$\begin{aligned} z_i t &= c_i e^{\omega_i^+ t} + \overline{c_i} e^{\omega_i^- t} \\ \omega_i^{\pm} &= \frac{-(\gamma\lambda_i + \eta) \pm \sqrt{(\gamma\lambda_i + \eta)^2 - 4\lambda_i}}{2} \end{aligned} \tag{3.34}$$

where $\omega_i^{\pm}$ represents the complex frequency for each mode. The real part of $\omega_i^{\pm}$ represents the damping of each mode and the imaginary part represents the (angular) frequency. The constant c_i is then found from the impact forces acting upon each particle, which is determined at run time, allowing us to compute for z_i and thus create our wave form.

3.2.3 Voxelization Methods

To overcome some of the limitations of existing tetrahedral methods, Picard et al. recently proposed a new finite element method for modal analysis to extract the modes [11]. This is done through the voxelization of the object. Instead of generating a tetrahedral mesh, the geometry is broken down into voxels and the system's matrices are created from the material parameters. The results, the authors claim, is a far more robust model, less likely to succumb to numerical instabilities.

The work is based on Nesme et al. [9], which uses hexahedral finite element for computing the mass and stiffness matrices of a mechanical system. They first generate a high resolution voxelization of the object, which are then merged to an arbitrary resolution. The mass and stiffness of a single voxel are then merged from its children, using a weighted average that takes into account the distribution of the material. To create the modal bank, a simple extension to the work of Nesme et al. is used to model the microscopic deformations for sound rendering, similar to O'Brien's method above. The rendered sound is then simply a sum of damped sinusoids, as in previous approaches.

The main benefits of such an approach are that it is more robust, and indeed more scalable. Tetrahedral meshes require a considerable amount of assumptions to be met in order to perform the modal analysis, most of which are inapplicable to hexahedral models. As a result, hexahedral analysis is much less likely to run into instabilities or difficulties. Secondly, it allows for greater scalability, as a single voxel can be created from the sum of its children, and vice versa. Not only can this be used to reduce the overall expense of creating the modal models, but it also allows for the future possibility of being able to scale the voxelisation at runtime, although this is dependent on improvements in hardware.

3.3 Using Recorded Data

One of the difficulties with the above methods are that the models can become very complex to assemble if the object is inhomogeneous, or indeed if the material parameters of the object are not well documented. For these reasons, van den Doel has tended to favour the creation of modal models through the use of recorded data [19][17][13].

Conceptually the process is very simple. A recording is made of an impulse striking the object at a particular location (which we shall denote k_i). By analysing this recording it is possible to identify the object's characteristic frequencies, and through their intensity we can identify the gains associated with that mode, and by watching how the intensity of each frequency changes over time we can identify the decay rate for the mode. Indeed, the process is so simple that it can be done by hand if preferred, needing only a spectogram for reference (which can be performed by almost any digital signal processing package).

However, if one wishes to assemble a high resolution model, automisation is necessary, as it would rapidly become a time consuming process. Van den Doel describes one such algorithm [19], using Windowed Fourier Transforms. The Fourier transform is a common method used in Digital Signal Processing to break up a complex time domain signal into the sum of its sinusoids. This allows us to describe the time domain signal into the sum of its frequencies (the frequency domain).

One of the limitations of a frequency domain representation of a signal though is that the output is not representative of any one point in time, but rather the duration of the signal that we have sampled. In order to be able to identify how the signal changes over time, it is necessary to perform multiple overlapped Fourier Transforms of the signal. And to avoid artifacts, we need to weight the signal by some window function, in this case a Hann window. The result of this is called a spectogram.

One interesting observation about modal synthesis is that any one mode tends to be represented as a peak on a spectogram. That is, any one mode tends to be represented strongly by one particular sinusoid and its most immediate neighbours to a lesser degree (often only one or two 'bins' away). It is this that we exploit when identifying the frequencies of the constituent modes in a signal. The algorithm itself is relatively simple. Once we have performed the Windowed Fourier Transform, we identify the peaks on the spectogram at any one point in time and vote for that frequency. Thus, the frequencies with the most votes are the modal frequencies we will use for synthesis.

Once the frequencies have been identified, it is then a simple matter of performing a linear regression as to how the intensity of each sinusoid decays over time. This is done by fitting the intensity to the linear equation $\alpha t + \beta$ (where t represents the next window in time). From α and β we can estimate the decay rates and amplitudes.

3.4 Radiation

An important observation about the above modal models is that they treat all modal models as a single point source. This is an important consideration because the deformations occur at various different locations throughout the object's geometry. Thus, the pressure waves are slightly convoluted when they reach us as sound, through diffraction and other artifacts of radiation. For example, if you tap a cup on the side, with one's ear over the opening, you will notice a significant difference in the sound than had you held your ear under the base.

In light of this, James et al. proposed an extension to the modal model to take into consideration the above phenomenon [7]. This is achieved by introducing the well documented acoustic transfer function to the synthesis pipeline. This allows for the effective modelling of diffraction and interreflection of soundwaves that affect the overall timbre of the object, resulting in a more realistic audio experience.

The method involves solving for each mode's acoustic transfer function $p(x)$ and introducing it to the synthesis pipeline in the form $p(x)e^{+i\omega t}$. $p(x)$ models the spatial component of each mode's pressure field, and is given in the form of a Helmholtz equation[1]. This can be a highly expensive process to solve, requiring many different boundary conditions to be taken into account. To allow for real time synthesis, they developed an estimator for $p(x)$ which utilises a precomputed estimations of the boundary conditions. This is what they term their PAT approximator.

Perceptually it represents a significant improvement to the modal model, although it does not take anything of the objects environment into account. However, it demonstrates the near endless possibilities and levels of sophistication the modal model can attain, reinforcing its importance to the future of interactive applications.

An alternative method for capturing the radiation of pressure waves in the modal model was proposed by van den Doel using his system 'Timbrefields' [13]. Instead of using numerically solved pressure fields, Doel uses recordings of an object's impulse response to capture the spatial variations of a sound relative to the listener. This is implemented by creating modal models from recordings of multiple contact locations as described above. However, instead of representing the amplitudes as two-dimensional matrix, they are represented by a five-dimensional vector that also takes into consideration the location of the listener. By also including impact recordings at various different listener locations, the gains are then found by interpolating the values found from the recordings.

Since this is performed as a pre-process, it is significantly less expensive than the above technique implemented by James et al., requiring only the interpolation between stored amplitudes at run time relative to the listener. However, the large amount of recordings that need to be taken require an automated system for doing so. Secondly, the five-dimensional representation of the amplitudes represents a significant increase in memory consumption.

3.5 Liquid Models

Until recently, physically based sound synthesis had only been applicable to the modelling of rigid bodies. However, there has been a number of attempts at creating

[1]the Helmholtz equation is the fourier transform of the wave equation

physically based models for the production of liquid sounds.

Van den Doel proposed an algorithm for generating liquid sounds based on bubbles [20]. The physics of bubbles become the focus. This is because water makes very little in the way of distinguishable sounds to a human. Only when air is trapped and reaches the surface are sounds perceived. This is because the surface tension excites the bubbles.

The physics of a bubble is relatively well documented. The impulse response of a bubble is the familiar sinusoid function, incorporating information on how both the frequency, dampening and gain change over time. It has been also found, that both the pitch and dampening constants are related to the radius of the bubble, thus only bubbles of a certain size produce perceptible noise to humans. And through an estimation of the energy maintained by a bubble, we are able to generate a prediction for the amplitude. From this, we can plug the values into any synthesis equation to create the bubble sound. One distinctive observation though about bubbles, is that their pitch tends to rise as they come closer to the surface of the water. This is because as the mass of water above decreases, less force is placed on the bubble's walls. From this, it is possible to create an estimation of how the frequency of the bubble changes over time.

For single drops, the model is unsatisfactory. Large bubbles sound unnatural, while smaller bubbles sound like drops on a hard surface. However, by exploiting the observation that bubbles are rarely heard in isolation, it is possible to use the model for certain sounds. To do this, van den Doel created a bubble generator that creates bubbles according to a probability distribution that emulates empirical observation of bubble creation in nature. It is found that by varying certain constants, such as the maximum depth, the bubble creation rate, and the radii, and the energy, it is possible to model certain sounds such as rain or waterfalls. It is also found that distance to the water source can be approximated easily by inserting an exponential constant.

Zheng and James developed a more physically based model [24]. They do so by expanding existing fluid based simulation for animation with particle systems that allow for the creation of bubbles. They then solve the acoustic transfer function for each bubble in the system (a very time consuming process). While it is far from a real time application, it is nevertheless a promising piece of work in the area of physically based sound synthesis.

There are four main steps to the process. They first simulate the fluid in question, and generate the corresponding bubbles. They then estimate the surface vibration of the fluid, and from this they estimate the pressure and vibration in the air, then render the sound.

The primary aspiration was for the application to work in conjunction with existing fluid models. In their implementation they used the Euler equations governing flow. They then simulate the bubbles as simple particles. The actual bubbles themselves can be modelled as a simple harmonic oscillator, as in van den Doel's work. Like van den Doel, bubble frequency is a function of time, however, their derivation is more physically based, taking into account the pace at which the distance from the surface changes. The acoustic radiation is then modelled using Helmholtz's equations, using a Green function to specify the boundary conditions.

In terms of computation, this requires to solve a Helmholtz PDE, for every bubble in the scene, for each time step. To do this they developed their own parallel integral solver, to approximate the Helmholtz-Green functions. The entire simulation itself is run on 96 separate cores, which is far from a real-time application. Nevertheless, the future remains bright. At the very least it demonstrates that the principles of physically based sound are not limited to rigid bodies.

3.6 Summary

This chapter has presented the main strategies for the creation of modal models. As shown above, the current state of the art allows for a wide variety of sounds and levels of sophistication, capable of meeting a huge range of the demands of current interactive applications. Secondly, all the models presented here can be automated which significantly reduces the burden of man-hours needed to create a highly sophisticated sound track for an interactive application. However, this is only one half of the picture. What remains is to model the forces generated by the rigid body simulator, and subsequently render the audio. This is discussed in the next two chapters.

Chapter 4

Modelling Impact Forces

For physically based sound synthesis, the creation of the modal model is only half the battle. In order to adequately synthesize contact sounds we still face the obstacle of resolving the resonator banks with the rigid body simulator. While our modal models have shown it is possible to decouple the rigid body motions from the surface deformations of an impact, we are still left with the difficulty of accounting for the differences between the visual frame-rate of the rigid body simulator and the audio frame-rate of the synthesis. In other words, we need some method to translate the impact forces of the rigid body simulator into forces suitable for the purposes of synthesis (which van den Doel terms the 'audio force'). This requires the need to extend the rigid body simulator in some manner to accommodate our modal model. However, how this problem can be approached depends greatly on how the modal model is derived.

4.1 Numerical Models

Using the models developed by O'Brien et al. and Raguvanshi et al., the only extensions required of the rigid body simulator are for it to retain and store the collision forces for the purposes of synthesis. Given a force vector **f** that describes the impulsive force acting on each vertex, it is transformed by the matrix that describes the eigenvectors of the system (G^{-1} in the case of Raguvanshi, or $V^T L^{-1}$ in the example of O'Brien et al.). This transforms the impulse into the local coordinates of the system.

$$\mathbf{g} = G^{-1}\mathbf{f} \tag{4.1}$$

Thus, the i'th entry of the transformed force $\boldsymbol{g}$ can be used to derive the amplitude of the i'th mode as below:

$$c_1 = \frac{2\Delta t g_i}{\omega_i^+ - \omega_i^-} \tag{4.2}$$

$$c_2 = \frac{2\Delta t g_i}{\omega_i^- - \omega_i^+} \tag{4.3}$$

in the case of O'Brien et al.'s model. Or for Raghuvanshi and Lin's:

$$c_i \leftarrow c_i + \frac{g_i}{m_i(\omega_i^+ - \omega_i^-)e^{\omega_i^+ t_0}} \tag{4.4}$$

both of which are then plugged directly into our synthesis algorithm at run time.

4.2 Models generated from Empirical data

For models created from recorded data, it is necessary to apply an additional layer of logic. The modal model generated by empirical data corresponds to $M = \{\mathbf{f}, \mathbf{d}, \mathbf{A}\}$. The amplitudes A are given by an $n \times k$ matrix, where n is the number of modes and k the number of contact locations. So each column of A corresponds to the observed amplitudes of an impulse response for a collision at location k. So in order to determine k, we need an estimation of the point of contact.

One such extension is described by Doel et al. [17]. To find an approximate contact location, they use sphere trees. They created the tree using a polyhedral approximation of their model, and then used the reported minimum distances from the sphere tree to compute the contact locations. A barycentric interpolation scheme determines the amplitudes from the nearest columns k in A. However, high resolution is required of both the trees and the recorded contact responses, otherwise artifacts are introduced.

Another difficulty arises from the decoupling of rigid body motions and surface deformations, with regards to the nature of the observed collisions. The reality is that due to the surface deformations of an object, even the simplest and hardest collisions are actually the result of a number of small short duration collisions between the two objects' surfaces. And secondly, it is still necessary to convert the dynamic force into the audio force.

A simple model for doing so is given by:

$$F_t = F_{max}(1 - \cos\frac{2\pi t}{T}) \qquad (4.5)$$

where F_{max} is the dynamic force generated by the rigid body simulator, and T is the total duration of the impact in terms of the audio frame rate. This is a non physically based model. However, it does seem to result in a good approximation. It rises slowly at first, and then rapidly, which is a good approximation of the elastic tendency of many materials. However, the suitability of it depends greatly on the type of impact being modelled. For example, it is noted in [17] that very hard collisions were better approximated by simulating a series of short duration impulse for the first few modes in an object.

4.3 Continuous Sounds

'Continuous sounds' is the term for impact sounds as the result of continuous contact between objects, namely rolling and scraping. By and large they are produced by 'irregularities' in the surfaces of two objects [17], and as such frequently the rigid body simulator will not provide enough information to allow for their adequate synthesis. Doel et al. demonstrated, however, that it is possible to use stochastic models to at least approximate these sounds.

They found scraping sounds are slightly better documented. They generate the audio-forces by first creating a surface profile of the objects in question. One requirement however, is that these profiles be created at a high enough resolution so that the surface can be sampled at the audio rate. The profile is created using a wavetable[1] generated from a 'roughness model'. Doel et al. generated this model from fractal noise passed through a reson filter[2]. Fractal noise was experimentally found to be a good indicator of roughness, and the filter scales the frequency according to the velocity of the objects. The wave table is then sampled using a linear interpolation algorithm.

As rolling is quite similar to scraping in the object's interactions, they were able to

[1]a wavetable refers to samples taken from a periodic waveform, or frequency spectrum, which can be used in a form of additive synthesis

[2]a reson filter is a recursive digital filter

extend the roughness model for rolling. What they observed was that rolling sounds seem to be characterised by much softer impacts which seems to excite only the lower frequency modes. So they implemented the roughness model with an additional low pass filter, to cut out higher frequencies. However, the results are not as effective as for scraping.

Stochastic models may not always be necessary for rolling sounds though. Raghuvanshi and Lin found that their technique did not need any additional handling beyond modelling the impulses [12]. This was largely due to their rigid body simulator, into which they had integrated more a more complex friction model. This resulted in much more accurate contact resolution, meaning the rigid body simulator alone was able to accurately model the softer impulses of a rolling object colliding.

4.4 Summary

While there are a number of different force models available in the literature for implementation, the task is not as well documented as the other components in the pipeine, and depends hugely on the models used. Most focus on modelling impacts as a series of small impulses. However, the success of this is determined both by the models used, and the overall friction model of the phyiscs engine. Indeed, we have noted that the most realistic demos in the literature have, in most cases, used custom extensions to the rigid body simulator. It shall be shown how the existing force models can be difficult to resolve with the off-the-shelf rigid body simulators in Chapter 7.

Chapter 5

Synthesis

Once the modal model has been created and the contact forces extracted from the rigid body simulator, what remains is to synthesise the sounds. As shown previously, the impact response of the modal model for an impulse can be described as:

$$y(t) = \sum_{n=1}^{N} a_{nk} e^{-d_n t} sin(2\pi f_n t) \tag{5.1}$$

Thus, to create the waveform, we must evaluate each modes individual response at each sample in time to create the waveform. This is referred to as time domain synthesis and requires careful handling. Large scenes with thousands of modes to be evaluated can cause a system to become quickly overloaded. It is therefore necessary to make the synthesis of the wave form as efficient as possible. The rest of this chapter describes a number of methods that have been developed to make the synthesis more efficient.

5.1 Time Domain Synthesis

As mentioned above, time domain synthesis is the process of evaluating each sample in time, in order to produce the wave form. However, this can be an exceptionally expensive process. Thus, this section explores the possible techniques to make time domain synthesis more efficient.

In order to see how this is possible, it is necessary to explore the form of the modal model again in more detail. An important point that thus far has largely been overlooked in this text, is that the modal model as originally described in Chapter 2 (reproduced here for clarity):

$$y_t = \sum_{n=1}^{N} a_{nk} e^{-d_n t} sin(2\pi f_n t) \tag{5.2}$$

is actually a simplification of the complex wave form:

$$y_t = \sum_n a_n e^{i\Omega_n t} \tag{5.3}$$

An alternative expression of the complex wave form is:

$$y_t = \sum_n a_n e^{i\Omega_n^+ t} + \overline{a_n} e^{i\Omega_n^- t} \tag{5.4}$$

where Ω_n^- is the conjugate of Ω_n^+. The important aspect to note is that the imaginary part of Ω_n describes the angular frequency of the mode (ω_n) and the real part describes the dampening parameter. This allows us to simplify equation 5.4 into the form of equation 5.2 through the observation that:

$$sin(\theta) = \frac{e^{i\theta} - e^{-i\theta}}{2i} \tag{5.5}$$

However, evaluating a sine function potentially forty thousand times a second for every single mode is still far too expensive a process for adequate real time synthesis.

Additional manipulations of the complex waveform can lead to some sizeable performance improvements. For example, van den Doel describes the process of transforming the complex waveform into a second order digital filter requiring only three floating point multiplies [18]. For the purposes of clarity, a brief summary of van den Doel's method follows:

The first step is to discretise equation 5.4, arriving at:

$$y(m) = \sum_{l=0}^{m} \sum_n a_n e^{i\frac{\Omega_n}{S_R}(m-l)} \tag{5.6}$$

This is a convolution equation. By exploiting its inherent linearity, it is possible to rewrite it as the recursion relation:

$$y_n(m) = e^{i\frac{\Omega n}{S_R}} y_n(m-1) + a_n \tag{5.7}$$

where m represents the sampled point in time, and S_R is the sample rate. This relation represents the audio signal Im(y), and requires only five multiplications to compute. Secondly it is always stable due to the exponential damping component.

Van den Doel further simplifies the equation by separating it into its real and imaginary components $(u(m) + iv(m))$, and writing it in the following form:

$$\begin{aligned} u(m) &= c_r u(m-1) - c_i v(m-1) + a \\ v(m) &= c_i u(m-1) + c_r v(m-1) + a \end{aligned} \tag{5.8}$$

where

$$\begin{aligned} c_r &= e^{-d/S_R} cos(\omega/S_R) \\ c_i &= e^{-d/S_R} cos(\omega/S_R) \end{aligned} \tag{5.9}$$

The real component can then be expressed in terms of the imaginary component:

$$u(m) = v(m+1)/c_i - c_r v(m)/c_i \tag{5.10}$$

which when substituted into Equation 5.8 yields:

$$v(m) = 2Rcos(\theta)v(m-1) - R^2(m-2) + Rsin(\theta)a(m-1) \tag{5.11}$$

where

$$\begin{aligned} R &= e^{-Im(\Omega)/S_R} \\ \theta &= Re(\Omega)/S_R \end{aligned} \tag{5.12}$$

The important thing to note is that $2Rcos(\theta)$, R^2 and $Rsin(\theta)$ can all be precomputed, allowing us to perform only three multiplications per sample. This is called a reson filter, (as it operates on its past inputs). This currently represents the peak efficiency for a time domain synthesis implementation, as currently described by the literature. It is not ideal though, as it requires evaluating every single sample for every single mode in a separate expression. However, it is more than adequate for all but the most demanding synthesis implementations.

5.2 Frequency Domain Synthesis

As opposed to constructing a digital filter, another method of synthesising collision sounds would be to use Inverse Fourier Transforms [1]. This is referred to as frequency domain synthesis, as it involves synthesising the output from the frequency domain representation of the signal (i.e. from the Fourier coefficients found using a Fourier transform). The advantages of this is that using a Fast Fourier transform algorithm (FFT) is much faster method to generate a signal. Where time domain synthesis requires the computation of each sample individually, using a single inverse Fourier Transform will create multiple samples for the specified window. This results in a more efficient operation.

One possibility for implementing this would be to create the Fourier Transform of an impulse response of a mode as a pre-processing stage, and then weighting these by the amplitudes of impact at run time, before performing an inverse FFT. However, while efficient, this would result in unacceptably high memory usage, as it would require storing multiple Fourier coefficients for each mode. To prevent this, Bonneel et al. [1] developed an efficient estimator of the Fourier coefficients for each mode.

They give the short time Fourier transform of a mode m for frequency λ at some interval t as:

$$\begin{aligned} s(\lambda) &\approx \frac{1}{2} e^{\alpha t_0} c_0 i \left(e^{-i\omega t_0} F_\lambda(H)(\lambda + \omega) - e^{i\omega t_0} F_\lambda(H)(\lambda - \omega) \right) \\ c_0 &= \frac{e^{-\alpha t_0} - e^{-\alpha(t_0 + \Delta t)}}{\alpha \Delta t} \end{aligned} \tag{5.13}$$

where $F_\lambda(H)(\lambda + \omega)$ is the Fourier Transform of the window function H (typically a Hann window, taken at the value $(\lambda + \omega)$.

The actual derivation of the above expression is quite involved and beyond the scope of this document. However, it is based on the simple observation that all the energy in a Fourier transform of a mode is to be found in a select few bins (Fourier Coefficients) around the characteristic frequency of that mode. Thus, the above expression need only be evaluated a few times per mode. Also, $F_\lambda(H)(\lambda+\omega)$ can be precomputed, and the complex exponentials are conjugate, meaning only one sine and cosine are needed. Overall the performance speed-up is approximately 5-8 times that of a recursive filter. On a modern dual core processor, they found that over 30,000 modes can be synthesised without artifacts, a significant improvement on previous implementations.

There are, however, two limitations with this approach. The first, is that the Fourier Transform limits what Bonneel et al. term the 'attack' of the sound, i.e. the very short burst of high frequencies on impact that characterise the timbre of the impact. This is rectified by the choice of the window function used in synthesis. It is typically found that the Hann window better encapsulates how a signal degrades over time, and results in smoother overlapping of frames. A rectangular window, however, better encapsulates a signal at a particular point in time. But this results in artifacts and additional resonance when used over the length of a signal. So the solution of Bonneel et al. was to use both.

The first synthesised frame is divided into four subframes. A rectangular window is used for the first subframe, and the square root of a Hann is used for the following 2, followed by a semi Hann to blend the rest of the sample into the next. All subsequent windows then use the square root of a Hann. The square root of a Hann is used to combat the second limitation, the audible stepping that can occur from FD synthesis. This is because each Fourier coefficient treats the signal as a point in time, and does not account for how the signal changes over the duration of the window. By using the square root of a Hann window twice, that is, once in the frequency domain to synthesize the signal, and once in the time domain to blend overlapping samples, these artifacts are eliminated.

5.3 GPGPU Acceleration

With the upsurge in GPGPU in recent years, there have been a number of attempts at utilising graphics hardware to accelerate the synthesis. This is allowed for the by the fact, as the modal model is simply a sum of a bank of damped harmonic oscillators, we can solve for each modes response individually, and then sum them to produce the buffer. This allows for the creation of a lightweight thread for each mode that would run on the GPU's shaders.

Zheng et al. first attempted this by implementing the synthesis as a fragment shader (before the advent of unified shaders and GPGPU APIs) [23]. The modal models were assembled into 2D textures, with each individual mode's frequency, dampenings and gains being stored in the RGB channels. The output was then rendered to a 2D render target.

The actual synthesis itself requires only two steps. The first pass computes the individual response for each mode (using van den Doel's reson filter described above). The second pass then sums the output for each mode into a single array for an object.

The observed speedup was quite significant, with up to 32,000 modes being calculated per second on a 256MB GPU. The same implementation on a single core CPU clocked at 2.8GHZ was limited at about 5,000 modes per second. However, the memory required to store each mode's input and output resulted in excessive use of bandwidth, and restricted the performance.

GPU acceleration remains a promising technique for making modal synthesis more viable for commercial applications, and recent developments such as CUDA and the creation of unified shaders have the potential to reduce some of the redundancy and memory latency observed in this particular application.

5.4 Perceptual based Speed-ups

5.4.1 Mode Compression

Another speedup technique was developed by Raghuvanshi and Lin. [12], which they referred to as 'mode compression'. This is derived from a very simple observation about auditory perception, that humans cannot distinguish between two distinct frequencies if they fall within a range. Thus, we can traverse the frequencies of a modal model, and if two frequencies fall between the imperceptible limit, they can be replaced by a single frequency.

The actual interval at which other frequencies cannot be distinguished is referred to as the Difference Limens to Change (DLC), and was first described by Sek and Moore in 1995 [14]. What they found was that frequency discrimination was a function of frequency. At 2KHz, the DLC is just under 5 Hz, meaning a frequency of 1998Hz is indistinguishable from a frequency of 2KHz. But at 8KHz, the interval is over 80Hz. What this means is that for any modal model, regardless of how many modes are originally found, the number of modes can be capped at no more than 1000, without much degradation on the resulting sound quality. Raghuvanshi et al. also note, that as most objects have a sparse frequency domain representation, there are typically no more than 200 modes needed to adequately synthesise the impact sounds. This represents a significant improvement in performance.

The actual implementation varies depending on the type of models used. As Raghuvanshi and Lin used the mass-springs method, they utilise the gains matrix (G, the matrix that represents the eigenvectors of the system). They simply sort through the gains matrix according to each column's corresponding frequency. Then for each set of frequencies within the DLC, they replace the corresponding columns with a single column equating to the sum of the discarded columns. This allows the model to maintain its physically correct responses, as the number of rows remains the same.

It can also be easily implemented for models made using recorded data. Like above, we simply traverse through the frequencies to find pairs of modes that fall within the imperceptible limit. For each pair, we then simply sum the amplitudes, average the dampings and replace the two modes with a single mode.

Modal compression is performed entirely as a pre-process, so there is absolutely no additional computation required at runtime. There is some perceptible loss in quality, particularly as two close frequencies produce phase variations that can be easily heard. However, overall the impact sound retains its original characteristics.

5.4.2 Mode Truncation

Mode truncation was also developed by Raghuvanshi and Lin [12]. The idea behind it is to 'stop mixing a mode as soon as its contribution falls below a certain threshold, τ'. This is based on the very simple observation that at the point of impact, there are a large number of high frequencies emitted that are characteristic of the object's timbre. However, after a very short period of time, their contribution becomes negligible and they fade away into the background leaving only a set of lower frequencies that characterise how the object resonates. Indeed they observe that after 0.2ms, typically up to 30% of an object's modes can be excluded. This represents a significant performance gain.

This is derived from the synthesis equation (3.34), which Raghuvanshi and Lin have manipulated to show the precise cut off point of a particular mode:

$$t^c \leq \frac{1}{-Re(\omega^+)} \ln\left(\frac{2|c||\omega^+|}{\tau}\right) \tag{5.14}$$

Given an impact, and the mode gain c, it is then simply a process of evaluating the above expression for each mode to ascertain its cutoff point t^c. Using the resulting value for t^c we can stop calculating the contribution of a mode when it falls below τ. The value τ itself is somewhat arbitrary. Raghuvanshi and Lin noted best results when $\tau = 2$.

5.4.3 Temporal Scheduling

Temporal scheduling was developed by Bonneel et al. [1]. While not a speed-up technique as such, it is a method designed to limit audio artifacts as a result of computational peaks. It was observed that most of the computational peaks that occur in the synthesis application arise immediately after impact. These peaks occur as a result of resolving the contact forces, and speed-up techniques, such as mode truncation, that require an evaluation on impact.

Bonneel et al. developed a simple method to combat this that exploits audio-visual asynchronicity. It is well documented that humans compensate unconsciously for delays in audio-visual synchronicity through causal inference. This means that if the associated audio of any audio-visual event is delayed, the discontinuity between the audio and visuals is not perceived, up to a certain threshold T (typically 200ms). Thus, if there are a large number of impacts that need processing, it is possible to delay the processing of certain impacts. For example, if there are 20 impacts proposed by the Physics engine, and 120 audio updates per second, it is possible to compute one impact per audio update without adversely affecting the end result to the user.

5.5 Quality Scaling

'Quality scaling', as coined by Raghuvanshi and Lin, is the process of scaling the audio quality of the scene to fit the computational budget. It is an essential component of any synthesis application as it is the final form of insurance against audio artifacts arising from excessive computational load. The requirements are simple; it must be efficiently implemented in the manner least obvious to the user.

In terms of modal synthesis, the only fully flexible technique is to place a cap on the total number of modes that can be evaluated at any one time. This was initially implemented by Raghuvanshi and Lin. [12]. For their implementation, they sorted the objects in the scene according to their position relative to the user, with the purpose of prioritising those objects in the foreground. The rationale is simple: objects in the foreground attract the attention of the user more than those at a distance. From this, the objects are assigned a computational limit, as defined by some arbitrary linear scale. They then evaluate the modes of each object according to its priority in line with the budget. The modes are then evaluated in order of amplitude, highest first, until the computational budget has been exhausted. If, however, the budget is not fully exhausted, the remainder is then allocated to the next object in the priority queue.

Quality scaling was then extended by Bonneel et al. [1]. Again, the observation on which their implementation is based is straightforward. They note that the overall energy of a sounding object plays an important role in perception. This is based on the observation that a large object impacting in the background will mask out the sounds of smaller objects in the foreground. For example, a piano impact at 50 metres will draw much more attention than a tin can falling at 5 metres.

To implement this, they compute the total energy for a given impact sound (E_s) and allocate the computational budget in line with this value. This is computed from the sum of each mode's individual energy (E_m), which is given by:

$$E_m = \frac{1}{4}\frac{\omega^2}{\alpha(\alpha^2 + \omega^2)}\left(a^2\right) \tag{5.15}$$

Modes are then sorted according to E_m. Thus the n modes evaluated are those with the highest energy. It is also efficient. The entire relation, with the exception of the amplitudes, can be precomputed.

5.6 Summary

Overall, the synthesis represents a different paradigm to that of the deformation and force models. Where the creation of the models is concerned with realism, the main priority for the synthesis is for it to be efficient. Currently, frequency domain synthesis represents the peak efficiency in the literature. However, there is great scope to improve on existing synthesis strategies, and thus enhance performance. These possibilities are discussed in Chapters 6 & 7.

Chapter 6

Design

The major components of a synthesis system have by now been outlined to the reader in full, and can be seen in Figure 2.2. However, there still remains a sizeable number of decisions to be made on the methods implemented. Firstly, we must decide on a suitable method for generating the modal models, which shall be selected from the methods outlined in Chapter 3. The resolution of the contact forces also requires some thought, although most of the decisions are determined by the nature of the components selected. Finally, how the synthesis is performed is crucial to the overall success and viability of the application. The rest of this chapter outlines the decisions that were taken, along with a number of contributions that were made to provide a more robust and efficient synthesis system.

6.1 Plan

The aspirations for this work are to provide a full and efficient synthesis pipeline that would be both easy to implement, yet sophisticated enough to use in a final end product. It was intended to develop techniques that would be both an improvement perceptually, yet a simplification in terms of implementation, to previous work. Thus, two main features were identified that this work would focus on:

1. Using modal models generated from recorded data to enhance the rendered quality of that from numerical models.

2. Creation of an efficient parallelisation strategy for synthesis.

3. Development of an efficient synthesis pipeline that was both robust and flexible.

The final results would then be demonstrated in a single interactive application (the details of which are discussed in Chapter 7).

6.2 Generating Modal Models

One of the major objectives for this project was to attempt to find a method of combining empirical and numerical models. It was felt this was an appropriate exercise, based on our observations of the demos made of other implementations. What was noticed, was that while numerical models tend to be accurate in the variation of frequencies depending on contact locations (their 'attack'), their overall 'timbre' is not quite as perceptually correct as those generated from empirical models. In short, we felt they sounded to be more 'toy like' in their timbre than those of recordings. We have no real rationale as to why this might be the case, perhaps the algorithms for recorded data are slightly more effective for compensating for nonlinearities in the deformations. However, we speculate that the discretization of the models might also results in a failure to adequatley capture the resonance of an object.

Our efforts were concentrated on finding a non physically based method that would achieve similar results. Specifically we wanted something that would be relatively simple to implement. Given what we already know about a modal model (specifically, that it is only the amplitudes of the frequencies that vary with impact), it might be possible to correct an FEM model by forcing the amplitudes of its natural frequencies to correspond to some minimum contribution as determined by a set of recorded models.

However, in order to evaluate the potential for this, it would first be necessary to decide upon the methods for creating modal models using both numerical methods and recorded data. This is discussed in the following sections.

6.2.1 Creating Numerical Models

As described in Chapter 3, there are a wide variety of different methods for creating numerical models. Initially, it was proposed to use tetrahedral methods, using a structural engineering application such as ABAQUS. However, it soon became clear that such methods were fraught with difficulty, both in parsing the geometry for the application to perform the analysis, and in the expertise required to perform such an analysis. FEM require extensive knowledge of vibrational and elastic theory, and great care is needed when constructing the tetrahedral elements to ensure they fit the assumptions of the analysis being performed. This represents a sizeable obstacle, both in terms of time and accessibility. And most audio programmers are unlikely to have the necessary backgrounds and resources to be able to fulfil this easily. As a result, simpler methods were felt to be more suitable in resolving the goals of this work. Thus, for the numerical models, Raghuvanshi and Lin's mass-spring simplification was selected [12].

6.2.2 An Algorithm for Recorded Data

With regards the algorithms for existing data, there was little to choose from. The only algorithm documented in full was that of van den Doel, as he presented it in his thesis [19]. As mentioned previously in Chapter 3, this algorithm identifies the peaks in the frequency spectrum to find the frequencies. What remains is to perform a linear regression of the intensity of the signal over time, in order to estimate the amplitudes and dampening parameters for each mode. For readers not familiar with the digital signal processing, we refer them to Appendix A, which gives an overview of the Fourier Transform and other concepts used here.

However, it becomes apparent that the algorithm is woefully ineffective at distinguishing those peaks created by noise from those that characterise our object's modal response. This causes the slightly bizarre phenomenon of the algorithm appearing to randomly sample the frequency spectra. This is because the algorithm traverses over each window of the Short Time Fourier Transform (STFT) of the recording, giving a vote to each frequency that it identifies as a spectral peak. However, it does not take into account the intensity of the signal, so noise can easily be overestimated in the results. Thus, it was necessary to devise a method to overcome this.

The solution is actually rather simple. By weighting each vote by the square of the intensity of the signal at that frequency, it is assured that the meaningful parts of the signal are easily identifiable from the noise. Thus, the algorithm we elected to use takes the following form:

1. Perform a Short Time Fourier Transform over the signal, of widow-size N and number of windows k.

2. Obtain the intensity of each window in the signal in order to determine the start of the impact. This is performed by taking the absolute value of the complex output of the STFT.

3. Initialise an array of bins of size $W/2+1$, which correspond to the output of the STFT. This will be used to store the weighted votes for each frequency.

4. Loop through the output of each window of the STFT. Create an inner loop to traverse each frequency and identify the peaks.

5. For every peak found, increase the value of bins(i) by the square of the intensity of frequency (i) in window k (i in this case refers to the i'th frequency of the STFT).

6. Select the bins with the largest values. These are the identified modes.

7. Regress the intensity of the signal onto time using a linear least squares algorithm, to obtain the value of the slope m_i and offset β.

The frequencies f, dampings d and amplitudes a are then obtained from the following relations:

$$
\begin{aligned}
f_i &= \frac{SR \times (i+t)}{N} && (6.1)\\
d_i &= \frac{Q \times SR \times m_i}{N} && (6.2)\\
a_i &= \frac{e^{\beta} \times \frac{d_i N}{SR}}{1 - e^{\frac{d_i N}{SR}}} && (6.3)
\end{aligned}
$$

This is largely unaltered from van den Doel's original algorithm, with the exception of step 5, which was modified to take into account the intensity of the signal. Another modification was to ignore finding the end of the signal. This was in the initial algorithm to ensure the regression was not being performed over empty portions of the signal, which would skew the damping parameters. However, in the presence of noise it can be difficult to obtain the end of the signal, and in general we found it more reliable to manually edit recordings to exclude empty portions.

The final alteration we made was to include a tuning parameter t when estimating f_i. There are more robust ways to estimate the frequency [17], however, for the purposes of this implementation it would suffice to simply tune the found frequencies to the correct pitch by hand by simply modifying the value of t.

Combining Recorded Models

One of the limitations of the algorithm above, is that it is a single modal model for a single contact location. It does not give any indication of how the amplitudes should change depending on the contact location of the impact. Thus, it is necessary to assemble a single modal model from multiple models of recorded data taken at various different contact locations, in order to capture how the amplitudes change depending on the location. Previous implementations have not discussed in detail how to perform this though, so we shall give a short account of our approach to the issue.

First the k modal models are assembled from their recordings (where k describes the number of contact locations allowed for, and thus number of recordings made). Taking the first models frequencies as the base, we then loop over all subsequent models to indentify the modes that share the same frequency. If a frequency of a subsequent recording is found that matches a frequency of the base, we enter the amplitudes into the corresponding columns of the amplitudes matrix, and take the average of the dampenings. It is a very simple method, but has been found quite effective.

One might argue that by discarding frequencies that are present in the base column, we are omitting crucial information on the model, however, we argue otherwise. This is because of the simple observation that frequencies not present across all the contact models tend be largely a result of noise. Certainly if one looks at a spectogram of two recordings of the same object struck in different locations, one observes that all the significant frequencies are present in both of the recordings to a degree, and the crucial information that defines the contact is in how their intensities change. Thus, no special handling is needed beyond what we mentioned above.

One clear weakness, however, is in the dampenings. Typically, averaging the dampenings tends to result in an overdamped model. An obvious possibility would be to take the mode of the dampenings for each contact location. However, in practice the least squares algorithm will never find an exact match. Through experimentation, we found the most perceptually correct method was to simply take the mean value, and if necessary scale it by same arbitrary constant C where $C \leq 1$ to correct them.

6.2.3 The Minimum Impulse Response

In order to evaluate this hypothesis, we first need to estimate this minimum contribution of a mode for any possible impact. This actually turned out to be a simple procedure. Given an assembled modal model from recorded data, we can assume that the minimum contribution of a mode m_i is its lowest observed amplitude in the ith row of the amplitudes matrix A. Thus we can construct what we term the 'minimum impulse response' of a modal model as $M_{min} = \{f_i, d_i, \min(A_i)\}$.

6.3 Parallelising the Synthesis

Modal synthesis is a highly parallelisable task, and indeed offers great scope for variety in implementation. For example, additive synthesis can be performed by constructing a portion of each sinusoid individually before adding them to the audio buffer. Or indeed, a different approach could be taken, where each sample could be solved individually of the others by evaluating the mode contributions at that particular sample. And based on the performance of previous GPGPU implementations, the potential performance gains from parallelisation are sizeable.

Initially, a potential implementation using CUDA was evaluated [27]. As opposed to the previous implementation by Zhang et al. [23], it was deemed the recent emergence if unified shaders might yield an additional performance improvement. This was primarily because unified shaders should allow greater scope in only placing parts of the synthesis on the graphics pipeline without impacting too severely on the graphics pipeline. Unified shaders are dynamically allocated the stages of the rendering pipeline to allow for great flexibility in the rendering pipeline. However, it soon became obvious from the documentation that the size of the small memory caches allowed per thread would largely negate any performance increase, and that computing large audio buffers could easily result in trashing, or at the very least, excessive use of bandwidth between passes.

Thus, we focused on parallelising the synthesis on the CPU, something previously not discussed in the literature. This was deemed an acceptable approach, as multi-core CPUs have become the norm throughout the industry, and further increases in processing power come largely from increased parallelisation (for example, Intel recently reintroduced hyper-threading on its Core i7 series, which results in 8 hardware threads per CPU). And as parallelisation practices have yet to become the norm in the Interactive Entertainment industry, there would be sufficient resources available to accommodate this in most interactive applications run on multi-core CPU architectures.

6.3.1 Selecting the Granularity

For a CPU implementation, the fine grained parallelism as seen in [23] would be unacceptable, due the much higher cost of launching a thread on the CPU as opposed to CPU. Also, it would introduce a much higher possibility for deadlock, as each individual thread of a modes response would ultimately be sharing the same resources (the audio buffer). Thus, a coarse-grained approach is deemed more suitable for an implementation on the CPU.

As the unit of parallelisation, the sounding object itself was deemed a good option. Each object's response is entirely independent of the other objects on screen. This is because only the contact forces are dependent on the interaction between objects, and these are determined by the rigid body simulator before the object is sounding. Thus, a coarse-grained approach can be easily constructed by simply threading each objects individual response.

Secondly, as mentioned above, when parallelising for the CPU, one must take into consideration the cost of executing the individual threads. While the GPU is designed with the purpose of executing numerous lightweight threads, the CPU is not. Thus, a fine-grained approach would be completely unsuitable for execution on the CPU, as the overhead of creating additional threads is sizeable (through informal experiments we found the additional cost incurred to completely outweigh the benefits of the parallelisation). A coarse-grained approach would perform better when threads are kept to a minimum. Thus we designed a strategy that would allow for flexible thread creation, so that as many threads could be created to exploit the nature of the current CPU, without overburdening the application.

6.3.2 The Strategy

The decision was taken to initiate all threads upon the initialisation of the synthesis application, based on the observations above. This would ensure that the overhead of thread creation would not impact on the cost of the real-time synthesis.

The unit of granularity would be the modal object, with the maximum number of possible threads being the total number of sounding objects on screen. This was selected as it allows for a high level of parallelisation and flexibility. Each modal objects response is entirely independent of the other objects on screen. This might seem at odds with perception, for example the sound of a ceramic cup falling on concrete is very different to it falling on wood. However, the response of the cup itself is only influenced by the collision forces between the two objects, which needs to be solved before the synthesis takes place, so the synthesis of both the cup and the wood can be performed entirely independently. This is the fundamental observation that the parallelisation is built upon.

Thus, the strategy is simple. An arbitrary number of modal objects are grouped together. A worker thread with its own continuous update loop is initiated on initialisation of the Sound Manager, and is assigned the group of objects to iterate over. Thus, the functionality of the thread would be to simply listen for flagged updates, and produce the buffer portions of its assigned objects that are sounding. As the collision forces are computed by the PhysX, the only additional requirements are to give the physics engine access to the modal objects so that a collision can be flagged. The worker thread will then simply see the new collision flag, and begin synthesising the audio upon its next update. Thus, in order to keep the rigid body simulator and the audio engine consistent, it is simply a case of using a number of Boolean flags in the objects themselves.

Dividing the workload among threads

While the above design showed a large capacity for potential performance gains, it is very much a naïve process. It assumes all threads have equal access to the same resources, and that all threads have an equal distribution of active objects. Thus, it is necessary to arrive at a more sophisticated budget allocation method in order to efficiently balance the workload among threads.

We decided on a method to do so, using Bonneel et al's energy estimator for each mode. The idea is simple, allocate objects to threads according to the resources available to it. To do so, we simply define an active working list for each thread and do the following:

1. Compute the maximum modes for each thread.
2. Compute the current percentage usage of a thread.
3. Allocate new impacts to the working list of the thread with the least usage.

Sounding objects then remain on the working list of a thread until they are no longer sounding. If a new impact occurs, we avoid reallocating an object and instead simply flag a new collision in the object itself, so the thread continues updating it as normal. In theory there is still a possibility of a single thread becoming overburdened by continually impacting objects. However, in practice, this no longer becomes a major concern, as this results in a relatively equal distribution of the objects amongst threads.

6.4 Quality Scaling

As discussed in the previous chapter, quality scaling is the process of degrading the overall quality of the audio by finding methods for excluding certain modes from computation. It is a necessary component for any synthesis application, as it ensure its robustness and extends the system's capacity to handle large amounts of impact sounds. The rest of this section describes the main design decisions for taken to implement quality scaling in this work.

6.4.1 Budget Allocation

In order to efficiently implement some of the perceptual speed-ups available, it is necessary to have an efficient and robust estimator of the budget allowed for each update. We devised a simple method to do this, that has no impact on the runtime burden of the synthesis. It involves simply performing a quick initial 'stress test' of the Sound Manager's capacity in the given environment, and takes no longer than a second to perform.

First, we create a new thread, and add objects to the total of 5000 modes to the scene. We then compute the maximum length of time allowed for each update (T_{max}) from the following:

$$T_{max} = \frac{SamplesPerUpdate}{SampleRate} \tag{6.4}$$

Using a timer accurate to the nearest nanosecond, we then record the time it takes to record an impact and synthesise five subsequent 'frames' of audio, and take its average. We then express this as a fraction of T_{max} and compute the maximum amount of modes allowed for by dividing the current amount of modes in the scene by the fraction of T_{max}.

$$\begin{aligned} F_{allowed} &= \frac{T_{average}}{T_{max}} & (6.5) \\ N_{max} &= \frac{N_{curr}}{F_{allowed}} \times C; & (6.6) \end{aligned}$$

It should be noted, that this is something of an optimistic estimator, and that larger amounts of modes may incur more latency than experienced in the Stress Test. However, in practice we found that by simply setting C to 0.9 in the above, we can be sure of the estimators robustness.

6.4.2 Energy Estimator

Using the above budget allocation, it would be possible to implement the energy estimator developed by Bonneel et al. (discussed in Chapter 5) in order to efficiently prioritise the modes for synthesis. Bonneel et al's method was deemed the most suitable as the only alternative in the literature is to use the square of the amplitudes [12]. The square of the amplitudes would be effective for numerically created models, however there is no gain in using it for models created from recorded data as they are already sorted in order of amplitude.

Two potential uses were determined for the energy estimator. The first would be in calculating the modal budget for each object. By calculating the total energy for a particular object it would then be possible to express this as a fraction of the total energy in the scene. This fraction can then be used to determine the maximum number of modes mixed for a modal object (n_{max}) from the overall maximum as determined by the stress test:

$$P_{allowed} = \frac{E_{object}}{E_{scene}} \tag{6.7}$$

$$n_{max} = N_{max} \times P_{allowed} \tag{6.8}$$

The second use would be in determining what modes in an object to prioritise, as in Bonneel et al. Upon a new impact, they simply estimate the energy for all modes in the object, and then sort the modes according to their energy. Thus, the next time the synthesis is performed the n_{max} modes that are calculated are those deemed to contribute the most according to their energy.

6.4.3 Mode Truncation

The final component we elected to implement was modal truncation, as developed by Raghuvanshi and Lin (discussed in Chapter 5). Mode truncation is the process of identifying when the total contribution of a mode falls below a certain level, and thus no longer mixing it in the main synthesis algorithm. As this implementation would focus primarily on the use of recorded models, it would be perfectly acceptable to implement this as a pre-process, leaving only the process of performing a single comparison for each mode at run time.

6.5 A Full Synthesis Pipeline

The final design for the synthesis pipeline can be viewed in Figure 6.1. To summarise, it is a simple two class design. Sound Manager is responsible for managing the buffers, allocating the computational budget, creating the threads and ensuring the Modal Objects are sounding in a state consistent with the visuals. The Modal Objects main functionality is to compute the buffer portions required for the playback, however, it also includes the functionality for computing the energy of a mode and its cut-off point. The design is simple, scalable and robust. The precise details of implementation are given in Chapter 7.

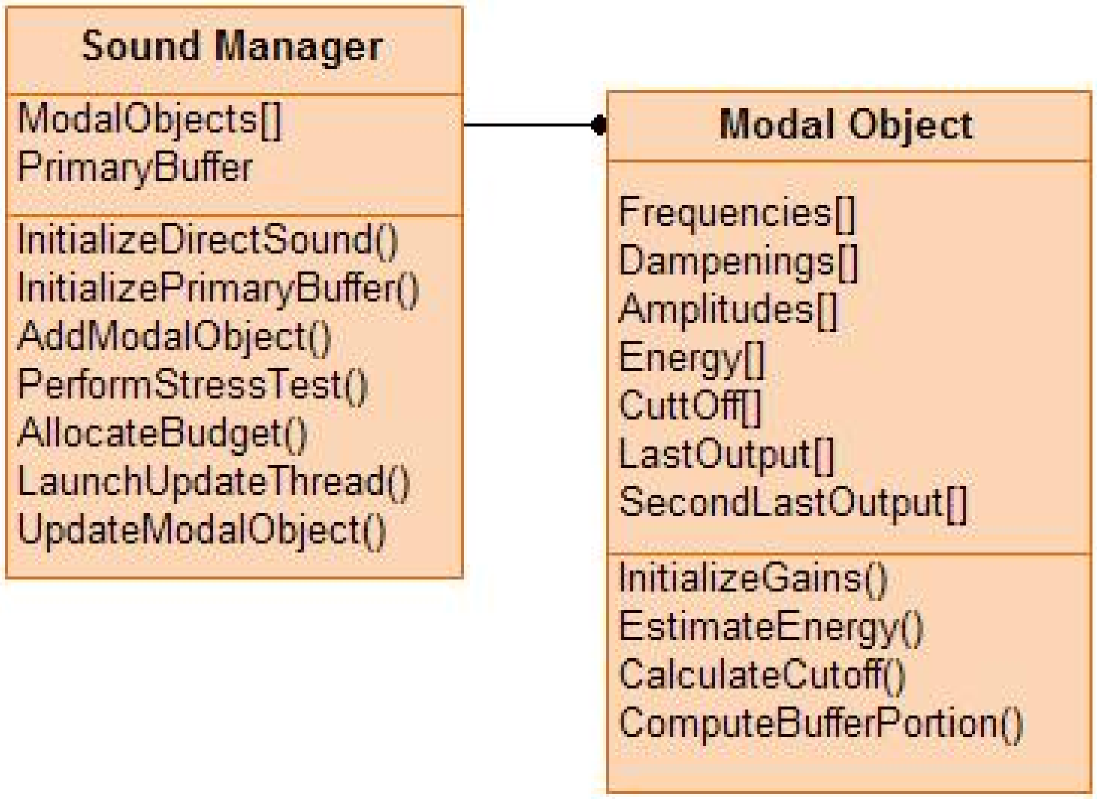


Figure 6.1: A class diagram illustrating the synthesis components

Chapter 7

Implementation

7.1 Component Choice

In line with the aspirations, we selected components that were widely available and portable.

7.1.1 Creation of the Models

The choice of algorithm for the creation of the models was the most difficult decision. As mentioned previously, Finite Element Methods are the most physically accurate, however, they are largely the most involving method for creating the models available. While conceptually simple, the large size of the systems of PDEs generated makes an implementation from scratch a daunting prospect. In the previous work it is common to see the modal analysis performed by using commercial applications such as Abaqus.

We elected to create the models using MATLAB [32]. This was something of a compromise, but nevertheless it is a product that is widely available in most academic institutions. And due to the simple nature of its programming syntax, code implemented in matlab is highly portable to other applications. It also has a number of routines such as LAPACK [31] and FFT that would simplify the process of creating models significantly.

7.1.2 Interactive Application

For the interactive application, there are a much wider variety of choices available for us to implement. The aspiration here was to use components that were freely available and well documented, that would reflect methods common throughout the industry.

Rendering Engine For the rendering engine Ogre3D was selected [34]. This choice was somewhat arbitrary, based on the large volume of documentation available for it and its frequent use in academia [1].

Physics Engine The Physics simulator to use was again somewhat arbitrary. It was decided to avoid using a simulator with custom extensions to model contact, as such modifications may not be feasible to implement in a highly demanding commercial application. Instead, it was intended to explore the complexity of linking physically based audio with existing simulators common in industry.

In the end PhysX was selected [33]. This came down to a number of factors; firstly, its GPGPU acceleration would free up more clock cycles for synthesis. Secondly, the documentation and level of support are quite good as it is a commercial application that has been used extensively in the industry. And lastly, that it has been shown to perform well for modal synthesis previously [1] [3].

Audio Playback With regards the audio playback, the choice was somewhat limited. We needed an API that would allow us not just playback audio, but directly manipulate and give us direct access to the audio buffers. In recent years audio APIs have focused much more on playback and positional audio (such as XAudio2), and as a result their scope has been limited when it comes to synthesis. We selected to use the DirectSound API [28], as it has been in existence for a number of years now, and is more comprehensive in its abilities than some of its newer successors.

As a side note, there were a number of SDKs available for synthesis, namely JASS [30], created by Van den Doel, and the Synthesis Toolkit (STK) [36], created by Perry Cook. However both are largely unsuitable for interactive applications, particularly JASS as it is written in Java. It was decided to avoid these as a result.

7.2 The Visuals

This section describes the process of integrating the rendering engine and the physics engine in order to create the basic framework for the interactive application. The following is simply a brief description of how the two components were integrated.

7.2.1 Ogre3D

Ogre3D is a comprehensive rendering engine. The SDK itself is encapsulated in the Root object. To render an object with Ogre, there are three main objects that need to be invoked. The first is the 'scene manager', which keeps track of all the objects on screen. Any object added to the scene manager takes the form of a 'scene node', with this node consisting of one or more 'entities', which define the geometry and other materials in the object.

A basic Ogre implementation was created using the tutorials and examples in the SDK. We used the ExampleApplication header file which initialises the SDK, and creates the SDK and the frame listener. The basic flow of the application is as follows:

1. Create the Root
2. Define the Resources
3. Choose the Render System
4. Create the Render Window
5. Initialise the Resources
6. Create the Scene
7. Create the Frame Listener
8. Begin the Render Loop

7.2.2 PhysX

The hierarchy of PhysX is surprisingly easy to work with, and a scene is built from a few small objects. The SDK itself is encapsulated in the class NxPhysicsSDK. Once initialised, this then returns a class NxScene which describes the physics environment and their interactions. A separate class, NxActor is used to initialise and describe each rigid body added to the scene.

The integration with Ogre3D is achieved in a few short steps, and was based heavily on an online tutorial [35]. First we initialise the PhysX SDK when Ogre creates the scene, and add our actors. Then we simply call the update in a frame listener declared using Ogre. The only additional handling required was to define a structure that encapsulates the PhysX actor and the Ogre node, to provide a one to one relationship when updating the node's world position based on that of the physics simulation.

7.2.3 Content Management

One other concern would be the coherent handling of geometry. For the application, a simple function was found that would allow us to use the Ogre mesh binary[1] and use the NxCooking Facility to create a corresponding triangular mesh for PhysX [35]. This should result in adequate coherency between both Ogre and PhysX.

As for the creation of the modal models, OBJ files were selected. This was due to their ease of parsing regardless of application or platform. The only concern is that the mesh binary and the OBJ file be coherent, however this is made less of an issue due to the abundance of different exporters available for most modelling applications. Coherency between the two would be a simple matter of choosing the correct exporter for geometry. For this implementation, we used a simple application called WinMeshView [37] to convert OBJ files into mesh binaries.

7.3 Pre-processing

As mentioned above, the creation of the modal models was implemented in a pre-processing stage using MATLAB scripts. Both van den Doel's algorithm for recorded data, and Raghuvanshi's mass spring methods were implemented in separate matlab functions.

[1]a .mesh file is a binary representation of a triangular mesh, and can be directly rendered in Ogre

Models from Recorded Data

In order to capture the recordings of impacts, we simply used a microphone from a Creative gaming headset. There was no elaborate method into creating the impulse responses, models were simply struck using a hard blunt instrument that did not resonate, typically the end of a plastic pen. The impacts were made as short as possible, and although it was attempted to be consistent with the force of impact, there were no specific measures taken to ensure this was the case. Audacity was used in order to record and edit the impulse responses [25]. Typically, one long recording was made of a series of impacts, and these were then edited and exported into a number of small wav files for analysis in MATLAB.

The algorithm was implemented with the extensions described in Chapter 6. Given a set of a recordings, we pass each of them through van den Doel's binning algorithm to get the set of frequencies, dampings and amplitudes that describe the impact. Once these have been obtained, they are assembled into a single modal model of the frequencies common across the recordings.

Numerical Models

For the numerical models, an implementation of Raghuvanshi's mass-spring method was created, again in MATLAB. The precise procedure is as follows:

1. Geometry is obtained by parsing OBJ files into MATLAB and stored as a structure.

2. From the vertex and edge descriptions, the corresponding mass-spring damping system as described in Raghuvanshi's algorithm.

3. The System matrices are then assembled from the mass spring system.

4. The solution to the system is found by diagonalising the elastic forces matrix, K, to obtain the eigenvectors and the eigenvalues.

5. The modal parameters are then obtained from the complex frequency of each mode which is found using the quadratic formula .

In order to speed up the development process, a number of functions were used from MATLAB Central[2]. Specifically, user-created files were used for parsing the OBJ files[3], and a second user-created matlab script was extended in order to assemble the system's matrices[4].

Formatting the Output

The greatest difficulty that arose during the development, was actually in attempt to find a simple way of formatting the output to be used in the interactive application. Initially, we created a specific filetype, .mm, to describe the different models, which we had hoped to parse from within the synthesis application. The format itself was simply a text file that contained a sequence of numbers to describe the frequencies, dampings and amplitudes, with an integer at the start of the file to indicate the total number of modes in the model. However, there were a number of downsides to this. It was found reading in the files was an exceptionally slow task that significantly added to the loading time. And secondly, the dynamic allocation of memory when the files were being read was extremely inefficient and resulted in a huge performance hit.

In the end, the simple process of directly generating C++ code in MATLAB was the most efficient. It was slightly cumbersome, as it meant having to manually add in the code to the project, however, in terms of performance it was by far the most effective solution, as it significantly reduced the overheads caused by the dynamic allocation of memory.

7.4 Real-Time Synthesis

The real-time synthesis system was developed largely by extending work done for a previous project [3]. This involved the two component design described in the previous chapter: A general 'Sound Manager' class responsible for the creation and maintenance of the audio buffers, including playback, and a separate 'Modal Object' class

[2]MATLAB Central is an online repository for sharing Matlab resources and community created scripts. It can be seen at: http://www.mathworks.com/matlabcentral/fileexchange/

[3]The specific function is 'readObj', created by Bernard Abayowa. It can be obtained at http://www.mathworks.com/matlabcentral/fileexchange/18957

[4]The function 'MDK' was obtained here, developed by Brent Lewis. It can be found at: http://www.mathworks.com/matlabcentral/fileexchange/14854

which would encapsulate all the data for a single sounding object, as well as including functionality to compute the audio samples. This design was ultimately similar to the previous project implemented [3], however, it was extensively rewritten. Indeed, the only aspect of the Synthesis system that remained entirely unchanged was the recursive reson-filter used to compute the audio samples, which was directly re-used from the original project, although like previously, it created using the DirectSound API. The rest of this section describes the development in detail:

7.4.1 DirectSound

The DirectSound API is an audio playback library that is part of Microsoft's DirectX framework [28]. It was designed primarily with WAV playback as its central theme. It is no longer actively maintained by Microsoft, as since DirectX 10, XAudio has become the main audio API of the DirectX system. However, XAudio is designed exclusively for WAV playback and its functionality for directly manipulating WAV data is limited, which is why the DirectSound API is still in common use, and indeed why it was selected for this project.

The central functionality of the DirectSound is accessed through its audio buffer features [8]. In DirectSound, there are two main types of audio buffer; the primary buffer and the secondary buffer. The primary buffer encapsulates the data that is directly played back through the hardware. The secondary buffer is then the object that the WAV data is directly written to. DirectSound will then mix all the active secondary buffers then directly to the primary buffer, which is then sent to the audio device.

The basic flow of a DirectSound implementation is as follows: An object, called 'device' that links to the audio hardware is initialised. Then a handler to the Primary buffer is created, although typically never used. Finally, any number of secondary buffers are created that can be loaded with data, and then simply triggered to play a sound effect.

For this application, DirectSound was implemented largely as described above. The device and primary buffer are both initialised upon creation of the sound manager object. Then for every modal object added to the scene, sound manager creates an additional secondary buffer that is stored in the modal object itself. Once an impact is detected, it is then a simple case of loading the secondary buffer with the audio output of the modal object, and then calling play on the secondary buffer, which results in the audio being mixed directly into the primary buffer and outputted. DirectSound provides plenty of additional functionality beyond this, in terms of special effects and 3D spatialisation of audio. However, for this implementation we avoided adding additional complexity to the audio engine to save both time and resources.

7.4.2 Synthesis Algorithm

The selection of the synthesis algorithm was made less by choice than necessity. Initially it was hoped to implement Frequency Domain synthesis, using the STFT estimator of a mode created by Bonneel et al [1] and described in Chapter 5. The initial aspirations were to implement the estimator, and then use the optimised Fast Fourier Transform library FFTW [29] to perform the inverse Fast Fourier Transform. However, there were a number of setbacks that prevented an implementation.

The first was that experiments with the estimator in MATLAB showed it to be highly unreliable. We were unable to ascertain whether this was due to an error in the implementation or due to a missing assumption as regards the nature of the Fourier Transforms performed. Either way, our results were entirely unstable. Occasionally the estimator would perform perfectly. Unfortunately more often than not it was so inaccurate as to hardly produce any audible output.

The second stumbling block that occurred was the failure to integrate the FFTW library into the interactive application. Again, it is not entirely clear as to why this would occur. However, we speculate it may be due to the fact the library was developed with UNIX systems in mind, and that perhaps some quirk of the Windows compiler or linker interferes with performance. Another possibility might be due to an incompatibility with Windows Vista. Regardless, given these obstacles, and a rapidly diminishing schedule, frequency domain synthesis was shelved, and we used a form of time domain synthesis that we had developed for a previous project [3].

The exact algorithm used was one developed by van den Doel [18] and described in Chapter 5. It takes the form of a recursive digital filter, where the output is some combination of the last two previous outputs, the exact combination being determined by constants derived from the frequency and dampenings of a mode. It is, to our knowledge, the most efficient form of time domain synthesis, needing only three multiplications to perform. Given the limited success with frequency domain synthesis, it seemed appropriate to reuse it for this application.

The algorithm itself is implemented in the Modal Object class. When an impact is triggered, the constants are initialised for each mode and stored in arrays. The Sound manager then simply requests subsequent buffer portions for playback, which are generated using the recursive filter. The last set of outputs are then continually stored in the object and reused until such point as the object is no longer sounding.

7.4.3 Parallelisation

The parallelisation of the synthesis was implemented using the Boost Threads Library [26]. The library largely acts as a wrapper for threads in C++, presenting a programming paradigm more similar to those familiar with Java or C#. Threads are launched by simply calling the 'boost::thread' function, and threads can be bound to specific functions using 'boost::bind'.

For this implementation, the parallelisation was performed in line with the design provided in Chapter 6. An additional update function was implemented in the Sound Manager object. This function would continually update the audio of a set of assigned modal objects, looping through each modal object assigned to it and creating the buffer portions as required. Once the Sound Manager was initialised, and all the modal objects added to it, the update function is bound to a number of boost threads, each with its own subset of the modal objects to iterate over.

The implementation of this was entirely straightforward due to the atomic nature of the modal object. As each modal object maintained its own data set to operate on along with its own buffer, there was absolutely no possibility of deadlock, or inconsistencies arising.

The only special handling required to ensure consistency was the processing of impacts. This was handled by the creation of a separate update thread with the sole purpose of processing impact data generated by the physics engine, which when it had processed new impacts, would simply activate a flag in the modal object to indicate a new impact had occurred and that audio needed to be synthesised. Overall, this led to an exceptionally lightweight, flexible and robust system, that could fulfil the requirements of the most demanding synthesis applications.

Dynamic Allocation of Workload

It should be noted, however, that the strategy to balance the workload among the worker threads was disabled in the final application. Overall, the budget creation and allocation is successful. However, occasionally floats so small that they are treated as zero are computed, particularly when calculating the energy of a mode. This has the undesirable effect of interfering with all the subsequent calculations and thus completely breaks down the budget allocation, and at times, completely eliminating the synthesis.

Thus, for the final implementation, we used the stress test to compute the maximum number of modes (described below). Then, the computational budget for an individual object is found from its total energy, and these allocated for every object in the scene. Thus to ensure effective parallelisation, we then simply distribute the workload evenly among threads regardless of whether the objects are sounding or not. In practice it introduces additional expense as the threads now iterate over objects not sounding. However, this additional computation does not severely impact on the overall rendered audio.

7.4.4 Speed-ups

The speed-ups selected for implementation (mode truncation, energy estimation and quality scaling) were discussed in Chapter 6. Implementation was uneventful, and performed exactly in line with how they were described in Chapter 5. As these were not the centre for the work, they are evaluated only briefly here.

Mode Truncation: This was implemented as described by Raghuvanshi. It is performed as a preprocess on initialization of the application. Overall it was found to have minimal impact on the overall cost of synthesis. In order for it to significantly reduce the computation it results in audible artifacts.

Energy Estimation: Again implemented as described by Bonneel et al. As we were using models from recorded data in the interactive application, it was implemented as a pre-process. It is performed immediately after the initialisation of the modal parameters. Overall it was found to be quite robust, although occasionally the calculations do break down as frequently the numbers computed are miniscule and treated as zero. It requires quite careful handling to implement.

Budget Allocation: As described in Chapter 6, budget allocation and quality scaling were performed using the energy estimator and the total time to compute a buffer portion. The stress test created to identify this was a simple implementation. A new thread is launched that simulates the impact and synthesis of 5000 modes for 5 frames of audio. The time taken is then averaged and used to compute the maximum number of modes. The modal budget is then allocated among the objects proportional to their total energy. Overall, it was found to be very effective and allowed a significant improvement to the total number of objects the synthesis pipeline could handle.

7.5 Resolution of Contact Forces

Most of the functionality for resolving the contact forces was provided by PhysX. This is achieved by extending the 'NxUserContactReport' class in the SDK.

NxContactNotify is a callback function that is called every time a collision is detected in the physics engine. The implementation is straightforward. First, when initialising the physics scene and adding the actors to it, we specified a number of flags that indicate we want notification of the collisions between the flagged objects. Thus, any time a collision is flagged, NxUserContactReport returns the pair of actors in contact and an event flag that indicates the type of contact. The necessary contact information is then provided by iterating over the pair using the 'NxContactStreamIterator' object in the SDK. This object then iterates over the manifolds of the actors in contact to find the locations of the contact, along with the force information. This information is then simply added to the object in question, and is used to call the audio synthesis in the next main update of the game loop.

What remains is to convert this dynamic force into an audio force; that is, process the force into a form suitable for audio synthesis. There are two steps to this. First, we must identify the vertex that has been struck in the collision. Secondly, we need to model the contact force as an impulse (or set of impulses).

Ultimately, we failed at the first step. It was planned to implement a sphere tree in order to resolve the contact location determined by PhysX with those accommodated for in the models. However, this stage was never reached. This was due to a number of factors. First, the modal models created were not particularly successful at capturing an object's variation (this is discussed in more detail in the next chapter). We simply did not have the hardware to make a modal model of recorded data at a high enough resolution, and the output of the numerical models was less than satisfactory. This meant we unable to validate whether an implemented technique had worked successfully or not. Secondly, the contact notification by PhysX did not appear particularly robust, and quite a few collisions went unflagged. We had initially shelved the idea of implementing a sphere tree until a successful workaround had been developed for the above. However, unfortunately time was exhausted before that stage was reached.

We did, however, successfully implement the second steps, modelling the impulses using the cosine function developed by van den Doel [18] and described in Chapter 4. The alternative was to model the impulses as a Dirac impulse of infinitely short time, as described by Raghuvanshi. However, this was found to be not as perceptually correct as van den Doel's model.

7.6 The Final Product

The final application is illustrated in Figure 7.1. This illustrates the breakdown of the main components into their major constituent classes.

The main game loop is maintained in the rendering engine. The rendering engine consists of two main classes: the main 'scene manager' into which all the geometry onscreen is added for rendering in the form of a 'scene node'. The second class is the 'frame listener', which unsurprisingly listens for a frame update of the main render loop and has a callback function that is invoked every time this occurs. This frame listener is arguably the most important component in the application, as it is the class that ensures the rendering, the physics and the audio are all in a consistent state.

The physics engine, again, has two main classes of note. The physics scene which describes all the actors that have been added and their current state in the physics simulation. And the 'Contact Report' class, which performs the processing of the collisions through invoking the ContactStreamIterator described above.

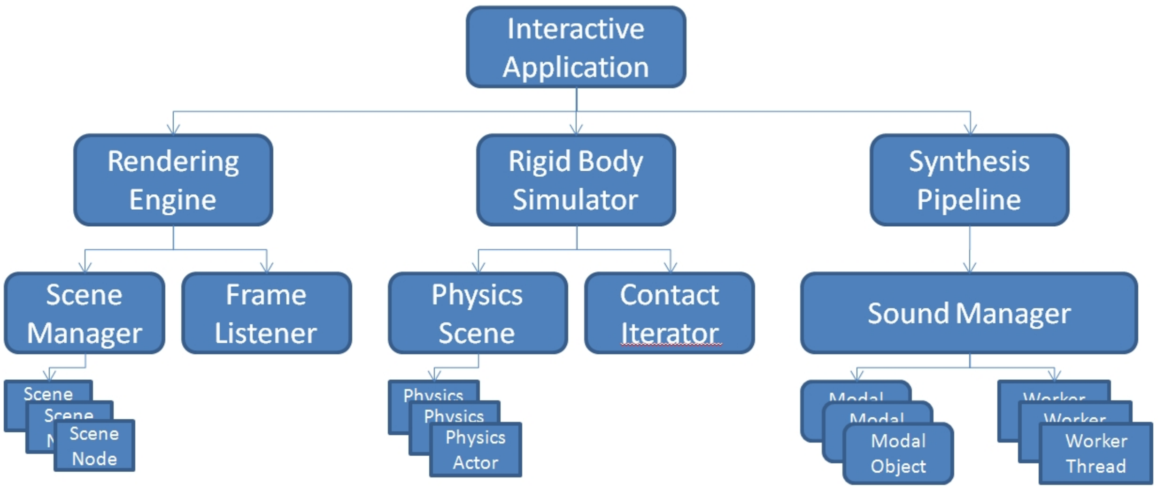


Figure 7.1: A high level class diagram of the interactive application

And finally, the Synthesis Pipeline, which consists of the sound manager, and any number of modal objects.

The overall flow of the application is as follows:

1. Initialise the Ogre SDK and load the content
2. Initialise the scene manager and the frame listener
3. From within the frame listener, initialize the physics engine and the audio manager
4. Add all geometry to the three components
5. Start the audio worker threads
6. Start the physics scene
7. Begin rendering

Then, every time the frame listener is called:

1. Update the positions of the scene nodes with the current positions determined by the physics scene

2. Flag and trigger the audio for any collisions that are detected
3. Advance the physics scene forward in time

7.7 Optimisation

In terms of optimisation, there was one specific task performed. This was to generate C++ code directly from the matlab scripts, as described previously. Overall, the application is not heavily optimised. Fixed sized arrays were used wherever possible, and did seem to render a slight performance improvement, however, for the most part, dynamically allocated arrays are necessary in most parts of the applications. One possible performance improvement might to use statically declared global arrays for the modal parameters, and use pointers within the modal objects to perform the synthesis. However most would shun thus as appalling design so it was not implemented here (although it should be noted, that a slight performance improvement was seen when using global variables during the development).

Chapter 8

Evaluation

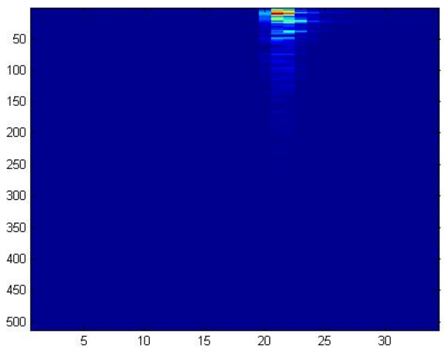
8.1 Creating Models

Models from Recorded Data

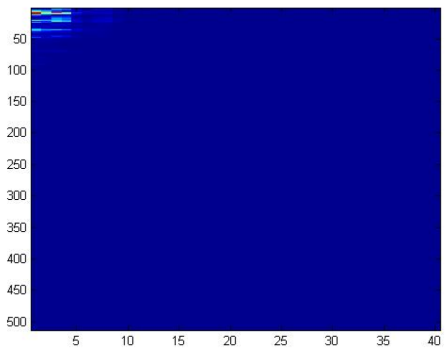
Success was had creating a variety of different models from recorded data. However, as was mentioned previously, the success of the recorded algorithms depends greatly on the type of material being modelled. This is illustrated in Figure 8.1.

Figure 8.1(a) shows the frequency spectrum of a recorded impact of a cardboard box. Figure 8.1(b) shows the impact response of the synthesised model. While in general the algorithm is very good at obtaining the dominant frequencies in the recording, it is clear that the resulting modal model is lacking, as beyond the $50th$ bin it fails to capture any of the frequencies in the resulting spectogram. The synthesized model is still very similar to the original model, and the distinguishing characteristics of the sound are preserved. However, when played side by side, there are notable differences in both pitch and timbre between the two models.

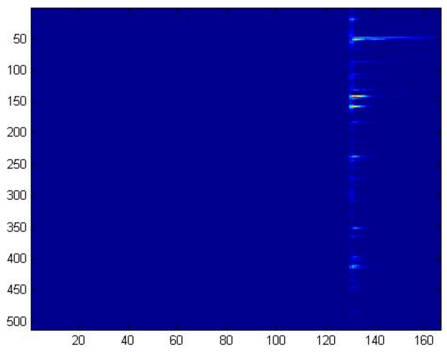
The same results were seen when modelling a ceramic cup. Figure 8.1(c) shows the recording, and Figure 8.1(d) shows the synthesised response. Overall the audio match is quite good, however again there is some discrepancy between the recording and the synthesis, primarily in pitch, although occasionally additional artifacts in phase are heard. While again the algorithm is only capable of picking up the dominant modes, it should be noted that the more equally distributed frequency spectrum of the ceramic cup results in a slightly better model.



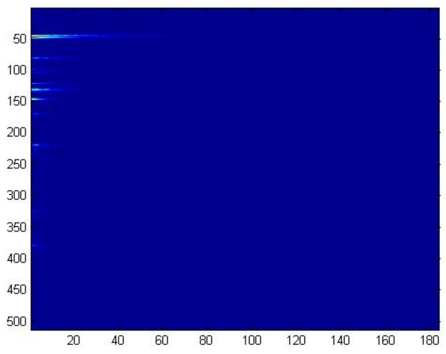
(a) The frequency spectrum of a recording of a cardboard box's impulse response



(b) The synthesised impulse response of a cardboard box



(c) The frequency spectrum of a recording of a ceramic cup's impulse response



(d) The synthesised impulse response of a ceramic cup

Figure 8.1: Spectograms illustrating the results of the empirical model

Additional difficulties were had, however, when combining the recorded models into a single modal model. This was primarily due to inconsistencies in the identified frequencies. Some models would be slightly above pitch, and others would be slightly below. There are more robust methods for identifying frequencies available than the algorithm implemented here, however, the problems are more due to the 'binning' process to identify the modes. If a mode's energy at a particular contact location is not strong enough to create an identifiable peak in the spectrum, it is generally lost in the noise of the signal which results in inconsistencies across the recorded models. This can be rectified somewhat by hand tuning each modal model, but by and large, there can be a significant loss in quality for certain contact locations. This suggests that algorithms for recorded data alone are simply not a viable solution in the long term.

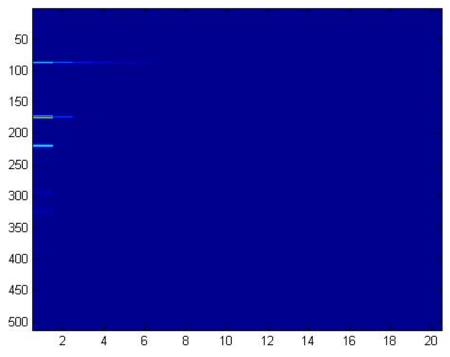
Numerical Models

The numerical models proved much more difficult, however, to extract meaningful data from than the recorded methods. Some success was achieved through the implementation of Raghuvanshi's mass-spring method. We successfully modelled an aluminium gourd, the frequency response of which can be viewed in Figures 8.2(a) and 8.2(c). It produces acceptable quality audio, along the lines of a metallic object. However, while its frequency response did respond correctly to excitation on exploring the frequency spectrum, the changes were too subtle to be heard in the resulting sounds without greatly exaggerating the input force data.

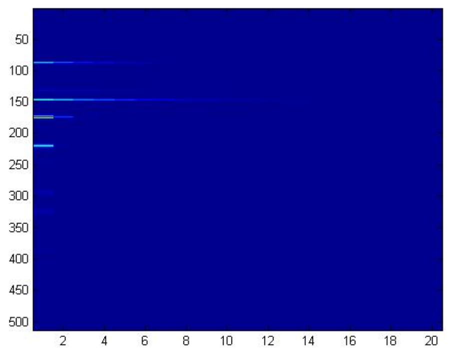
Combining Recorded Data with Numerical Models

Unfortunately, limited success was attained when generating other models through these means. This is primarily because the correct values for the fluid and damping parameters were not well documented in the literature. This is a major obstacle that needs to be overcome in order to create an effective synthesis pipeline for a large scale application, as currently, it is difficult to model objects with enough variety to create a truly immersive experience.

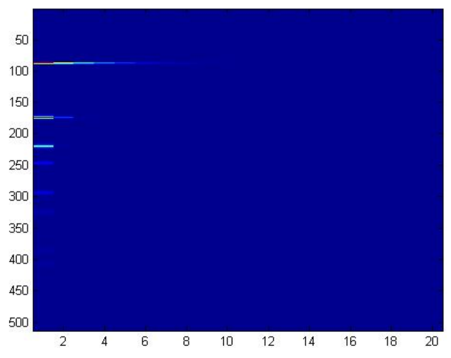
Greater success was had when combining the minimum impulse response with the numerical model. The results of this can be viewed in Figure 8.2. The minimimum



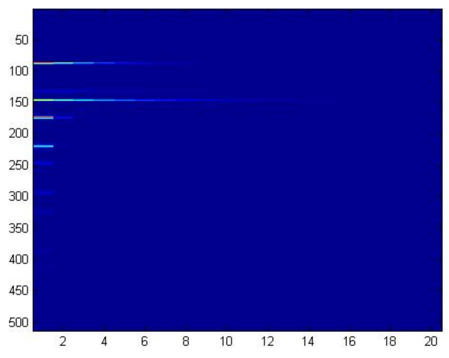
(a) The numerical model alone for an impact on the 2nd vertex



(b) The numerical model and minimum impulse response combined for an impact on the 2nd vertex



(c) The numerical model alone for an impact on the 144th vertex



(d) The numerical model and minimum impulse response combined for an impact on the 144th vertex

Figure 8.2: Spectograms show the result of combining numerical and empirical models

response captured was that of a ceramic object, and this response was overlaid on top of the response of the metallic gourd discussed above. Overall, the impact on the audio was quite successful. The minimum impulse response successfully added a ceramic feel to the resulting rendered audio, however, not to the extent of drowning out the original impact response and masking its subtle variations on impact. Overall, we feel this has promise in overcoming the limitations of the numerical models.

8.2 Performance

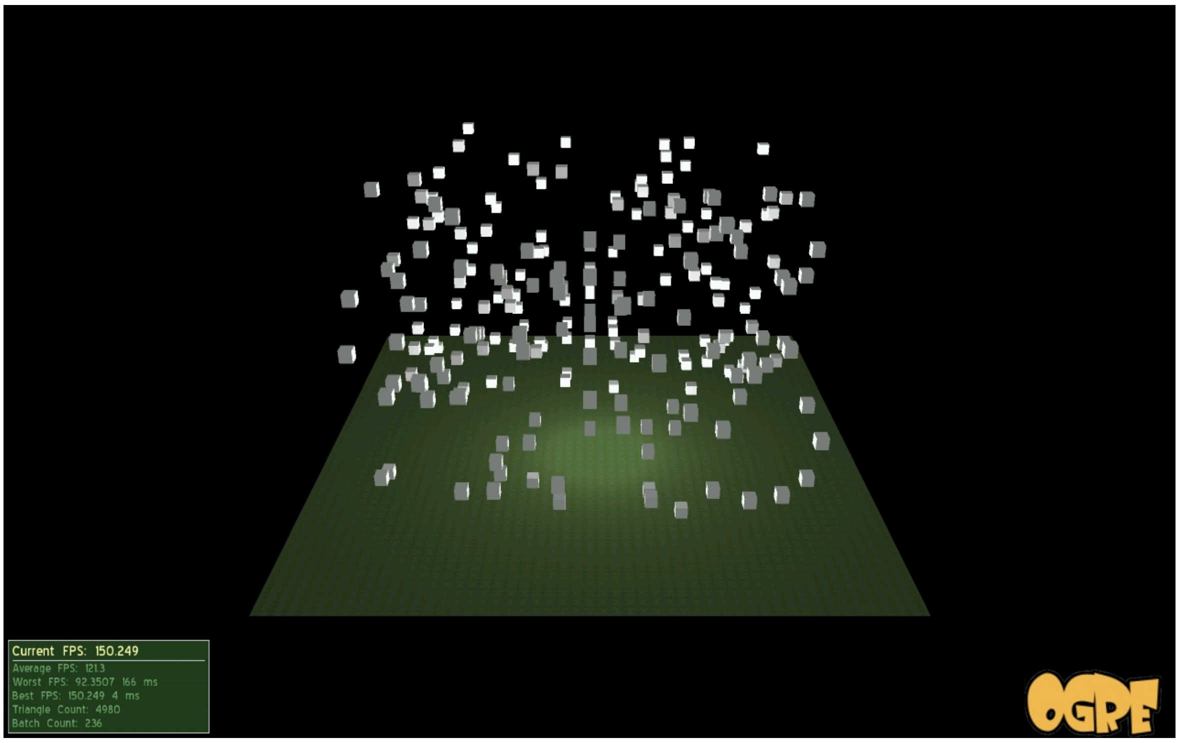
Performance was tested on a Core i7 920 quad core CPU clocked at 2.66GHz with an Nvidia GTX285 GPU.

8.2.1 Test Scene

In order to test the performance of the interactive application, we created a number of simple test scenes. We modelled a number of boxes using the recording technique described previously. Each box contains 50 modes, and was modelled as either ceramic, or cardboard. We then placed an arbitrary number of these objects on screen to model their collision with a simple plane. The plane itself is not sounding in the test scenes. The rendering and audio creation were implemented in separate threads, and the rigid body simulation was accelerated on the GPU. The test scene itself can be seen in Figure 8.3.

8.2.2 Synthesis

The overall performance is listed in Table 8.1 for a variety of different test scenes. Overall the application was capable of handling approximately ten thousand active modes with ease. Above this, notable artifacts are introduced to the audio quality, although the application itself does not break down entirely until we added over 400 hundred objects, which also represented the limits of the rigid body simulator. This represents a somewhat disappointing figure, although there are no adequate comparisons to be made as the literature generally does not reveal the total number of active modes at any one point in time. However, while the numbers were disappointing, the synthe-



```
Current FPS: 150.249
Average FPS: 121.3
Worst FPS: 92.3507 166 ms
Best FPS: 150.249 4 ms
Triangle Count: 4980
Batch Count: 236
```

Figure 8.3: Screenshot of the developed application

Number of Objects	Total Modes	FPS	Perceived Audio Quality
100	5000	190	Good
196	9800	145	Some asynchronicity
225	11250	120	Some asynchronicity
298	14450	100	Noteable artifacting
400	20000	60	Severe Distortion

Table 8.1: A table illustrating the performance of the application under different configurations

Number of Threads:	1	2	3	4	5	6
T:	2.14s	0.86s	0.464s	0.2997s	0.2478s	0.086s
$F_{allowed}$:	42.81	17.21	9.2822	5.993	4.9566	1.7393

Table 8.2: A table illustrating the performance of the application utilising different levels of parallelisation

sis system has demonstrated itself capable of handling all but the most demanding of interactive environments.

Effectiveness of the Parallelisation

In order to evaluate the parallelisation, and additional test was performed. As opposed to the previous perceptual test, the focus of this would be the maximum time taken to perform a single update. The strategy was simple, the total time taken T would be expressed as a fraction of the maximum allowed time for a single update $F_{allowed} = T/T_{max}$. Thus, if this figure became greater than one, the application was no longer synthesising sounds in real time.

A single test was performed, where 1225 sounding objects were placed on screen. As budget allocation was disabled, this would mean a total of 61,250 modes that had to be rendered simultaneously. The cost being evaluated was that of computing a buffer portion of 2205 samples, which was to be rendered at 44100Hz. Thus the maximum time allowed for any update was no more than 50ms, in order for the audio to be rendered in real-time. The results are given in Table 8.2.

The results are as one would expect. Overall the synthesis scales especially well using our coarse-grained approach. The comparison of the single thread test as opposed to the six threaded shows a massive improvement in performance, with an overall capacity greater than twenty times that of the single threaded application.

However, there are some questions raised by these findings. In particular, why is that the numbers show the multi-threaded application with a modal capacity greater than thirty thousand modes, whereas the informal perceptual tests showed the application breaking down at just over ten thousand? It is not entirely clear, however, it should be noted that there was no handling introduced should the computed audio buffer be of a higher bit rate than that allowed for by the direct sound implementation (it's normally not an issue for WAV playback and not documented). Thus, it could easily be the case that at approximately ten thousand modes our output becomes too loud for the audio device to adequately render.

8.3 Contact Sounds

The quality of the resulting variations in contact sounds were mixed. Successful contact modelling was somewhat hampered by PhysX's contact notification structures. By and large, they are successful. However, they do not appear particularly robust, and occasionally impacts go unflagged. Handling the resulting forces for audio was also not particular successful, largely because PhysX's contact forces seem greatly exaggerated, it frequently outputs values higher than 10^{10}. Scaling these forces by an arbitrary value was an obvious necessity, and seemed successful. However, very large forces are occasionally still flagged, which can frequently cause distortions in the output of the reson filter. A simple resolution would be to place a cap on the maximum value. However, this itself is undesirable as it results in the impacts realism being reduced. Using a higher bit rate might rectify this, however, this will result in higher memory and computational times.

Another limitation, was the failure to introduce models that adequately captured an object's sounding variation in response to different contact locations. This partly due to the weaknesses discussed above, and also partly due to the difficulty in resolving the contact locations as flagged by the rigid body simulator with the modal models. This is again a process not particularly well documented in the literature. It was hoped to implement a sphere tree, however, this would still represent a significant additional computational load to the overall application.

8.4 Limitations and Weaknesses

Aside from the limitations discussed above, the primary weakness of the system was its robustness. In theory, having dynamic budget allocation of the synthesis workload should prevent artifacts and distortions, even if the resulting audio suffers a dramatic loss in quality. However, in practice this is a very difficult task to achieve. This is primarily due to additional computation of the impact processing and resolution of the force data. Through test results, we found computing the initial buffer portion of an object's impact was on average twice as expensive as producing all the subsequent audio portions (60ms to create the first audio frame of 5000 modes as opposed to 30-35ms for all subsequent frames in our above test scene). Thus, large numbers of impacts at any one point in time will break down all but the most sophisticated of synthesis solutions. Efficient strategies for resolving the impact forces are something, we believe, requires more extensive research if commercial applications of physically based sound are to be computationally viable.

Chapter 9

Conclusion

9.1 Achievements

We were successful in implementing a full synthesis pipeline and achieved many of the initial goals we had set out for this work, although they were not quite up to the level of sophistication we had sought from the onset. Nevertheless, the main achievements are listed as follows:

1. Successful creation of modal parameters from recorded data, including some amendments to the existing algorithms to make them more robust.
2. Creation of a useful strategy for combining numerical models and models from recorded data in an inexpensive manner.
3. Somewhat successful contact resolution and extraction of forces from the rigid body simulator (although it still requires extensive work).
4. Design of an efficient parallelised synthesis pipeline using a flexible coarse-grained approach for multi-core CPUs.
5. Implementation of a successful strategy for dynamic budget allocation.

Overall, the viability of Physically Based Sound was explored and discussed. In short, a successful implementation of a full pipeline has been produced in a relatively short space of time, through limited resources. This demonstrates the viability of the techniques for a full scale commercial implementation. There is extensive additional research that needs to be carried out (some of which will be discussed below), however, the current state of the art is not far off the production of a final end product. Ultimately, it is felt this work has given additional strength to the foundations of physically based sound and has provided additional insight into the creation of a full synthesis pipeline.

9.2 Future Work

Accessibility of the Area

One of the main difficulties experienced throughout this work was the process of locating and assimilating the main concerns in the literature. This is primarily because the literature is quite fragmented and somewhat limited in scope. This makes it difficult to identify the key works in the area and more importantly, to relate them to each other. As such, there is a distinct lack of introductory material or standard works in the area, and this represents a significant barrier to anyone wishing to engage with the topic.

In light of this, we believe much more work should be made to provide a single standardised framework for the implementation of Physically Based Sound. It is our belief that this work is one of the few that attempt to combine all of the major contributions so far into a single volume of work. If Physically Based Sound is ever to have widespread commercial appeal, standardised methods and accessible tools will be essential to providing any kind of viable coherent framework. Thus, one of our main aspirations would be to create something of an international forum for Physically Based Sound, in the hope of providing a dedicated platform for promoting and sharing ideas across the community.

Creation of Modal Models

The creation of modal models is also another area that we feel requires continuing research. As stated previously, it was felt that Finite Element Methods, while no doubt effective, were cumbersome to work with, particularly when it came to resolving the input geometry with that which is to be rendered. Complex geometry can be highly expensive to perform modal analysis on, and without careful handling, the risk of running out of memory is considerable.

It is our belief that the future lies in combining numerically created models with models created from recorded data. However, considerable effort is still required for this to be achieved. There is a small body of research in the area of structural engineering that focuses on correcting the numerically created models with experimental data [16]. It should be possible to provide a framework for applying this to the area of physically based sound. However, this also requires additional extensive research with regards making methods for identifying modal parameters from recorded data more robust and reliable. Nevertheless, we have shown there is sufficient data captured in recorded models already to allow us to enhance the numerically created models, if we so wish. In future we would hope to provide a more physically correct solution for doing this.

Frequency Domain Synthesis

Frequency domain synthesis remains one of the most promising techniques to efficiently synthesise the impact sounds. However, attempts to implement it in this application proved fruitless. However, we found the estimation of the STFT to be unreliable. Our attempted implementation estimated some modes with uncanny accuracy, however estimations of other modes would be so far removed from the correct values that they would hardly produce an audible contribution. That is not say we consider the technique itself to be unreliable, more that the documentation of it seems incomplete and not fully developed. We believe more research is required to ensure its reliability and confirm its effectiveness.

Another aspiration would be to parallelise the process using GPGPU techniques. Frequency domain synthesis is a much better candidate for hardware acceleration than time domain synthesis, primarily as summing the STFT estimation of all the modes prior to the inverse FFT synthesis greatly reduces the required bandwidth. Secondly, the FFT itself is known to be highly parallelisable, and there are dedicated APIs for this purpose already in existence. A reliable implementation of the STFT estimator, coupled with hardware acceleration of the inverse FFT should be capable of synthesising more sounding objects than most existing rigid body simulators can handle. Thus, frequency domain is one of the prime candidates for further research within the existing pipeline.

9.3 Final Remarks

At the time of writing, and given the resources available, it is believed this work is among the most holistic attempts undertaken in the area of Physically Based Sound since van den Doel's original thesis. Through the analysis, and explorations, the current state of the art was deemed to be something of a mixed affair. Current modal models are robust and perform well, however, they do not seem to be as accurate as one would expect. Current force models, while effective, are not particularly well developed, and perhaps lacking in implementation details. And the synthesis, while no doubt efficient, is still slightly more expensive than one would like. Overall, the literature was found to be a collection of elegant ideas but somewhat lacking in a coherent strategy and methodology.

Through the achievements in consolidating the existing volume of work, clear strategies were developed for a more effective synthesis pipeline. And while the work is far from complete, it shows significant scope for improvement. However, while those who work with the traditional audio production techniques struggle to find new advances in realism, Phyiscally Based Sound has already shown itself to be a viable alternative, offering better realism, and reduced man-hours. In time, one hopes that these techniques will find their way into the industry and fully develop, so they may lay the foundations for a truly immersive interactive experience.

Appendix A

Digital Signal Processing

Digital Signal Processing is the term used to describe a family of methods to analyse, process and generate signals [15]. When a periodoc signal is converted into a digital form, it is represented as a series of samples of varying values. The process of generating meaning from these values (which will often seem highly random in nature) is the main focus of digital signal processing.

A.1 Pulse Code Modulation

Pulse Code Modulation (PCM) is the dominant technique for representing and creating audio signals in a digital format. The magnitude of an audio signal is sampled uniformly, and then quantized to allow it to be stored in a numeric form.

One vitally important aspect is the rate at which the audio is sampled (sample rate), as this is hugely determinant of the overall audio quality. The most important principle is what is known as 'Nyquist's law'. This states that in order for a sound to be captured digitally, it must be sampled at least twice the value of the maximum frequency [4]. As humans typically can hear frequencies up to a maximum of 22,000Hz, most audio is sampled at 44,100Hz.

A.2 Digital Filters

A digital filter is a system that operates on streams of data in a uniform manner [4]. A digital filter will typically operate on both its past inputs and outputs. A simple digital filter is given as follows:

$$y_n = \alpha x_n + y_{n-1} \tag{A.1}$$

where x is the input, y the output and n is the current sample in time. Digital filters are used extensively in audio processing, for adding effects, removing noise and frequently for analysing the input signal also.

A.3 Fourier Analysis

The Fourier Transform is a family of methods, named after Jean Baptist Fourier (1768-1830) [15]. Fourier claimed that any periodic signal could be represented as the sum of a select few sinusoids, and while the notion was soundly rejected at the time, nevertheless it led to the creation of Fourier Analysis, which is fundamental to signal processing today. The purpose of Fourier Analysis is to represent a complex signal in terms of simple sinusoids, which are easier to infer meaning from.

There are four separate families of Fourier Transform. The Fourier Transform and the Fourier Series are used for signals that are continuous and infinite in length, and as such are impossible to perform using digital computers. The Discrete Time Fourier Transform (DTFT) is used for discrete signals that are aperiodic, and the Discrete Fourier Transform for signals that are periodic and discrete. However, as the DTFT assumes a signal of infinite length, the only possibility for discrete signals is to use the DFT. However, to do so, it is necessary to assume that our **aperiodic** digital signal is simply a single period of a **periodic** signal.

A DFT transforms any signal from its **time domain** representation to its **frequency domain** representation. The time domain is so-called because the most common form of input signal is through 'samples taken at 'regular intervals of time' [15], often denoted by $x[]$. The frequency domain is then used to describe the representation of the signal in terms of its sines and cosines and their amplitudes, $X[]$. A forward

DFT is used to transform a time domain signal into a frequency domain signal, and the inverse DFT transforms a frequency domain representation into a time domain representation. Another important note is that $X[]$ is two parts, both real and imaginary, both of length (N/2+1). This is because the DFT can only discern frequencies up to one half of the length of samples the DFT is performed upon. The actual set of samples a DFT is performed upon is frequently referred to as **the window**.

A.4 Short Time Fourier Transform

The most common application of a DFT is in the area of spectral analysis. One of the common variations is to use the **Short Time Fourier Transform**. This is where a time domain signal is broken down into a number of different windows, of equal length, to perform a DFT on each window. This allows us to analyse how the frequencies and amplitudes of a signal change over time. It is typically graphically represented as a spectogram. When performing spectral analysis it is also common to see each window multiplied by a window function, (most commonly a Hamming or Hann window). This is because window functions tend to smooth out the signal, and reduce the level of noise in the resulting spectrum.

A.5 Fast Fourier Transform

The DFT can be expensive to compute, and as such it is especially important for synthesis applications to have an optimised implementation. The most common algorithm used is the **Fast Fourier Transform** (FFT). The FFT is a highly complicated algorithm, based on the complex DFT. In short, it decomposes a signal of N points into N signals of a single point. For each signal it then calculates the corresponding frequency spectra, and then synthesizes these into a single frequency spectrum. In terms of speed, for synthesis it would typically be in the area of 300 times faster than all other implementations (assuming a 1024 point window) [15].

A.6 Complex Frequency

The complex frequency, typically denoted by ω, is a representation of sinusoids used extensively in DSP. For the purposes of Physically Based Sound synthesis the imaginary part of the waveform ($Im(\omega)$) denotes the natural frequency in radians per second, and the real part ($Re(\omega)$) represents the value of the exponential dampening. However, more commonly used in DSP are the phase, represented by the angle between $Re(\omega)$ and $Im(\omega)$, and the magnitude of the signal, given by $|\omega|$.

Appendix B

Additional Resources

B.1 Source

The accompanying CD contains the source code of the implemented pipeline.

Interactive Application The folder *Application* contains the developed interactive application along with the necessary libraries to run it.

Empirical Model The folder *Recorded* contains the matlab script that implements the algorithm for creating recorded models. It also contains a number of recordings for the reader to experiment with.

Numerical Model The folder *Numerical* contains the implementation of Raghuvanshi's mass spring method, along with a number of obj files for the reader to experiment with the algorithm.

Miscellaneous The folder *Miscellaneous* contains another of other matlab scripts the reader may find useful, including the attempted implementation of the STFT estimator.

B.2 Demo

A demo of the developed application can be found online at:
http://www.youtube.com/watch?v=hJTR4rR5278

Bibliography

[1] N. Bonneel, G. Drettakis, N. Tsingos, I. Viaud-Delmon, and D. James. Fast modal sounds with scalable frequncy-domain synthesis. *ACM Transactions on Graphics (SIGGRAPH Conference Proceedings)*, 27(3), August 2008.

[2] Rob Bridgett. Designing for next-gen game audio, 2007. http://www.develop-online.net/features/65/Designing-for-Next-Gen-Game-Audio.

[3] Benen Cahill. Physically based sound synthesis, 2009. Unpublished CS7033 Project.

[4] Perry R. Cook. *Real Sound Synthesis for Interactive Applications.* A. K. Peters, 2001.

[5] Robder D. Cook. *Finite Element Modeling for Stress Analysis.* Wiley, 1995.

[6] Jimin He and Zhi-Fang Fu. *Modal Analysis.* Butterworth-Heinemann, 2001.

[7] D. L. James, J. Barbaric, and D. K. Pai. Precomputed acoustic transfer: Output-sensitive, accurate sound generation for geometrically complex vibration sources. *ACM Transactions on Graphics (ACM SIGGRAPH)*, 25(3):987–995, July 2006.

[8] Mason McCuskey. *Beginning Game Audio Programming.* Muska & Lipman / Premier-Trade, 2003.

[9] Matthieu Nesme, Yohan Payan, and François Faure. Animating shapes at arbitrary resolution with non-uniform stiffness. In *Eurographics Workshop in Virtual Reality Interaction and Physical Simulation (VRIPHYS)*, November 2006.

[10] J. F. O'Brien, C. Shen, and C. M. Gatchalian. Synthesizing sounds from rigid body simulations. In *ACM SIGGRAPH Symp. on Computer Animation*, pages 175–181, 2002.

[11] Cécile Picard, François Faure, George Drettakis, and Paul G. Kry. A robust and multi-scale modal analysis for sound synthesis. In *Proceedings of the International Conference on Digital Audio Effects*, September 2009.

[12] N. Raghuvanshi and M. C. Lin. Interactive sound synthesis for large scale virtual environments. In *ACM SIGGRAPH Symp. on Interactive 3D Graphics and Games (I3D)*, pages 101–108, 2006.

[13] John E. Lloyd Richard Corbett, Kees van den Doel and Wolfgang Heidrich. Timbrefields — 3d interactive sound models for real-time audio. *Presence: Teleoperators and Virtual Environments*, 16(6):643–654, 2007.

[14] Aleksander Sek and Brian C J Moore. Frequency discrimination as a function of frequency, measured in several ways. *Journal of the Acoustical Society of America*, 97, 1995.

[15] Steven W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical, 1997.

[16] A. Teughels, J. Maeck, and G. De Roeck. A finite element model updating method using experimental modal parameters applied on a railway bridge. In *7th International Conference on Computer Aided Optimum Design of Structures*, pages 97–106, May 2001.

[17] K. van den Doel, P. G. Kry, and D. K. Pai. Foleyautomatic: Physically-based sound effects for interactive simulation and animation. In *Proceedings of SIGGRAPH 2001*, pages 537–544. ACM Press / ACM SIGGRAPH, 2001.

[18] K. van den Doel and D. K. Pai. Audio synthesis for vibrating objects. In *Audio Anecdotes*, 2003.

[19] Kees van den Doel. Unpublished doctoral thesis, 1998. http://www.cs.ubc.ca/ kvdoel/pubs.html.

[20] Kees van den Doel. Physically-based models for liquid sounds. *ACM Transactions on Applied Perception*, 2(4):534–546, October 2005.

[21] Kees van den Doel and Dinesh K. Pai. Synthesis of shape dependent sounds with physical modeling. In *Proceedings of the International Conference on Auditory Display*, 1996.

[22] Kees van den Doel and Dinesh K. Pai. The sounds of physical shapes. *Presence: Teleoperators and Virtual Environments*, 7(4):382–395, 1998.

[23] Qiong Zhang, Lu Ye, and Zhigeng Pan. Physically-based sound synthesis on gpus. In *Entertainment Computing, ICEC 2005*, 2005.

[24] Changxi Zheng and Doug L. James. Harmonic fluids. *ACM Transaction on Graphics (ACM SIGGRAPH)*, 28(3), August 2009.

[25] Audacity: The free, cross-platform sound editor. http://audacity.sourceforge.net/.

[26] Boost c++ libraries. http://www.boost.org/.

[27] Compute unified device architecture (cuda).
http://www.nvidia.com/object/cuda_home.html.

[28] Direct sound audio api.
http://msdn.microsoft.com/en-us/library/bb219818%28VS.85%29.aspx.

[29] Fastest fourier transform in the west (fftw). http://www.fftw.org/.

[30] Java audio synthesis sytem (jass). http://people.cs.ubc.ca/ kvdoel/jass/.

[31] Lapack - linear algebra package. http://www.netlib.org/lapack/.

[32] Matlab. http://www.mathworks.com/.

[33] Nvidia physx game physics solutions.
http://developer.nvidia.com/object/physx.html.

[34] Ogre3d rendering engine. http://www.ogre3d.org/.

[35] Physx & ogre3d integration guide (in russian although sourcecode is provided). http://www.ogre3d.ru/wik/pmwiki.php?n=PhysXAmpOgre3d.IntegrationGuide.

[36] Synthesis toolkit in c++ (stk). http://ccrma.stanford.edu/software/stk/.

[37] Winmeshview. http://techhouse.brown.edu/ dmorris/projects/winmeshview/.