Real-time Physically Based Audio Generation

Graham McCann

M.Sc. in Computing

Institute of Technology Blanchardstown

2003

Supervisor: Hugh McCabe

Abstract

Using physical simulation to control movement and interactions between objects in a 3D environment, such as a computer game, has become increasingly common. However, this idea has not been extended in a general sense to the audio domain. The audio component of these systems usually comprises of pre-recorded sound files that are triggered in response to events. We argue that this approach has serious limitations and propose instead that the physical information made available by physics engines provides an opportunity to carry out synthesis of appropriate sounds from scratch. We outline a framework for adding a sound synthesis module to a physics engine and discuss some approaches to implementing this.

Acknowledgements

I would like to thank my thesis supervisor, Hugh McCabe, for his active participation in this research and his constant support. I would also like to thank my family, my fellow post grads and the staff at the Institute of Technology Blanchardstown. Finally, I would like to thank Steve Collins and Havok for providing their physical simulation software for use in this project.

GRAHAM MCCANN

Institute of Technology Blanchardstown
March 2004

Contents

Table of Figures

Chapter 1

Introduction

1.1 Background

Over the past decade many techniques developed by the academic community have found a practical and lucrative application within the games industry. In particular, solutions to problems found in the fields of computer graphics and artificial intelligence (AI) have proved extremely useful to games developers. With the ever increasing capabilities of modern computer hardware, particularly micro-processors and 3D graphics accelerators, it has become possible to do more complex calculations on the fly, greatly increasing the physical realism of computer games in the last few years. A number of companies have surfaced in recent years developing real-time physical simulators that can be integrated with existing graphics engines to create a visually believable virtual world. Most of the systems in use today use a technique called *rigid body simulation* (Baraff 1993, 1998; Witkin et al, 1990). This form of physical simulation involves modelling objects as solid geometric shapes, each having basic physical properties like mass, inertia, and friction. These objects are then placed in a virtual world where the normal laws of physics apply, such as gravity and momentum. At a specified interval, in general at least once per screen frame update, all the forces in the virtual world are resolved and the objects moved accordingly. Because the geometries of the bodies are constant, i.e. non-deformable, it becomes possible to perform rapid physical simulation in real time without having too

great an impact on system resources. This removes the need for animators to either script every possible action in the game or restrict the range of actions that can be done in the game, and instead gives rise to more dynamic, spontaneous and interactive forms of animation and gameplay. The recent wave of computer games that incorporates this form of simulation have only used them for small components of the system, such as projectile weapons and limited amounts of moveable scenery, but the new generation of PC and console games are expected to be using rigid body simulation to a much greater extent, creating worlds that contain a much greater level of interactivity and physical accuracy than previously, e.g. Max Payne 2, Hitman 2.

Figure 1.1 – *Example of use of Havok physics engine in use in a modern game. Image taken from Max Payne 2.*

Having a visually impressive system is of obvious benefit on its own, but in order to create a truly immersive experience for the user it is necessary to seamlessly integrate both vision and sound (Preece et al, 2000). If the sounds do not match the corresponding elements of the visual aspect of the simulation, the illusion of realism will be lost, no matter how impressive that visual aspect may be. Academic research into technologies which may be applicable to games development has a tendency to stay a few steps ahead

of the technologies used in the mainstream gaming industry. The primary focus of developers in the last decade has been on graphical realism, whereas progress in audio areas has been relatively slow. One of the likely reasons for this is the hardware industry's tendency to focus on the improvement of graphics hardware technologies, such as pixel-shaders and full-screen anti-aliasing whereas the audio hardware has had relatively few additions, the EAX technology from Creative Labs being one of the only major advancements. The current practice for computer games and virtual environments is to use pre-recorded sounds, known as samples, for the audio components of the system. This method has hardly changed its fundamental approach at all in the last decade, but with good reason; it has the advantage of requiring a minimal number of processor cycles to play back the samples, since they can usually be played back without the use of CPU consuming devices, such as advanced filters. The simplicity of this method also gives rise to a number of disadvantages, including:

?? The difficulty involved on the part of the developers of having to gather all the necessary sounds together before the game is completed.
?? The repetitive nature of the pre-recorded sounds can eventually detract from the realism of the system, particularly if they are used quite often over an extended period of time.

The sample gathering that a team will inevitably have to do during development of a game can become quite an arduous task if the game is quite large and contains a vast array of sounds. Also, if the game requires the use of sounds that are rarely heard in games, or maybe have not been used before at all, then the task of finding these sounds or creating them convincingly becomes an even greater one. There are sample collections available to developers, such as the BBC Sound Effects Libraries or the Universal Studios SFX Library, which provide a broad spectrum of royalty-free sound effects for use in all forms of entertainment production. While products like these can greatly reduce the time spent on sample gathering, they too present their own set of problems. Since these collections will undoubtedly have been purchased by many companies in the entertainment industry there is a high likelihood that some of the same samples will end

up being used in more than one project. While the user may accept a sound not being exactly like what it is supposed to be on screen in one game, if they hear the same one again in a slightly different situation it will remind them that they are just playing a game, rather then being immersed in a virtual world.

The repetition of sounds is a noticeable problem even within the same environment. When a user has to listen to the same gun-shot or the same foot-step continuously when they are playing the game it can become more of an annoyance than an immersive experience. Due to sampled sounds being static recordings on the disk or in memory, each time they are played they sound exactly the same as the last time, or even the last hundred times. In traditional real-time animation the simulated collisions, from an audio perspective at least, are a simple matter of what objects collided with each other and they do not usually take into account the specifics of the collision. In reality the exact points of impact on the colliding objects as well as the magnitude of the impact force are very important in determining the resultant positions after the collision. The same factors come into play when determining what sounds can be heard. Take for example, two cubes hitting together. If they were to collide face to face (i.e. flat surfaces together) a particular distinctive sound is produced. On the other hand, if one of the blocks was tilted at an angle so that one of its corners hits one of the faces on the other block, a moderately different sound is produced. The difference in sound can sometimes be only very subtle, but still significant enough that the user would notice.

The limitations of sample based sound systems are only confirmed by the inclusion of physical simulation in the environments. For all the visual benefits that the physical realism, specifically rigid body simulation, can provide for game-play, it also presents a new set of problems that sample based methods would find difficult to solve. For one thing, the huge variety of situations that can exist as a result of using rigid body simulation would each require an appropriate sound, most of which could not be known without extensive testing of the physically simulated world. Conversely, traditional systems have only a finite set of pre-scripted events, so the necessary sounds can easily be determined and obtained beforehand. Also, the improved realism in the visual aspects

of a game will require the audio to move up to the same level to keep the user immersed in the virtual environment. As stated earlier, both the visual and audio aspects of the system need to be of equivalent quality for full immersion for the user, or a game can be a disappointing experience due to the inferiority of one of these elements. The repetition of samples, as previously described, will become even more evident in a situation like this, so a suitable alternative is required at this stage.

With the inclusion of physics engines in many modern games however, the foundation is already there for the addition of *sound synthesis*. In short, sound synthesis is a way of creating sounds through mathematical algorithms as opposed to just playing back pre-recorded sounds stored in memory. The physics engine already calculates a considerable amount of data for its own purposes, and would usually just discard any superfluous information once it has achieved the movements required. Instead of disposing of the intermediate physical calculations we can use this data to assist in the synthesis of an accompanying effects soundtrack. As previously mentioned, rigid body simulation is the most common form of physics simulation. We will now describe this in more detail in order to give the reader some understanding of how it can assist in the sound synthesis process.

1.2 Rigid Body Simulation

1.2.1 Fundamentals of physical simulation

Physical simulation means that the traditional methods of specifying movement by key-framed animation are done away with and dynamic methods are used in their place. This was first introduced in the academic community by researchers such as (Baraff, 1995; Witkin et al, 1990; Hahn, 1988; Moore & Wilhelms, 1988). The fundamental idea is that the laws of classical mechanics can be used to compute the movements of objects within a simulated world. Most important of these are Newton's Laws of Motion, which

provide the building blocks upon which the rest of the classical mechanics formulae are derived. These laws describe the relationship of motion to the forces that cause it. The three laws are as follows:

1) Newton's first law states that a body will remain in its current state, either stationary or in motion, until a force is exerted upon it, thus giving the object acceleration.
2) The second law specifies that the vector sum of all the forces acting on a body is equal to the body's mass times its acceleration. This gives rise to the well-known equation:

$$F ? ma \qquad (1.1)$$

   where F is the net resultant force on the body, m is the rest-mass, and a is the acceleration.
3) The third law states that if a body exerts a force on a second body, then the second body will exert a force with the same magnitude, but opposite direction onto the first body. This law is more commonly known by the phrase, 'for every action there is an equal, but opposite, reaction.'

Although Newton's laws and any formulae derived from them are the primary sources of data for a physical simulation, they are not the only ones. Other fundamental laws of classical mechanics, such as the conservation of momentum and the conservation of energy play a significant role in the simulation, with these two conservation laws being of particular relevance to rigid body simulation. When these laws are applied to the virtual world they can be used to determine how the forces that are applied to objects should affect the behaviour of these objects in the world. For instance, when an object is at a particular position in the air at a moment in time and a force is applied to it for that moment, then a proportional acceleration is applied to the body which will determine its movement over the next few steps of the simulation (Newton's second law). The effects of gravity are a perfect example of this in a normal simulation. Likewise, by using the

conservation of momentum law as well we can determine what resultant force is applied to each body after collisions and consequently what accelerations to apply.

In summary, rigid body simulation is based on the idea that if we know an object's mass and we also know what forces are acting on it at a particular moment in time (these could be environmental forces such as gravity or forces that have resulted from collisions with other objects), then we can compute how that object should move.

1.2.2 Implementing the physics engine

Now that the laws of classical mechanics have been established we can examine how they can be applied to produce realistic physical simulations on screen. The underlying principles of rigid body simulation operate independently of either the graphics or audio components, existing as a purely mathematical representation of the environment. In order to perform physical simulation, a representation of the physical world is required. For each item in the environment a polygonal mesh will already have been created to serve the visual aspects of the game. For the purposes of the physical simulation a second geometry is required, which in most cases will be a simplified version of the visual one. While the graphical models of objects in the game may be very detailed, e.g. a player character having spikes on a suit of armour or the inclusion of wing mirrors on a car, these additional cosmetic features would only add needless additional strain onto the physics engine. Instead, each rigid body in the system is assigned one of the simpler secondary geometries (or more than one if the objects are complex enough, or the object will be divided during the game, e.g. breaking or smashing) and also provided with properties like mass, friction, elasticity, etc. At regular intervals (often at a higher rate than the graphics are displayed) a call is made to the physics engine requesting an update on all the objects in the scene. All the elements in the environment are analysed, specifically any forces or impulses which may be acting on objects in the scene, and the resulting positions and orientations of the items in the environment are calculated. The graphics engine can then update the visual geometry as it renders each frame to reflect

this. The simulation loop for an application incorporating a physics engine would resemble something like the diagram below (figure 1.2):

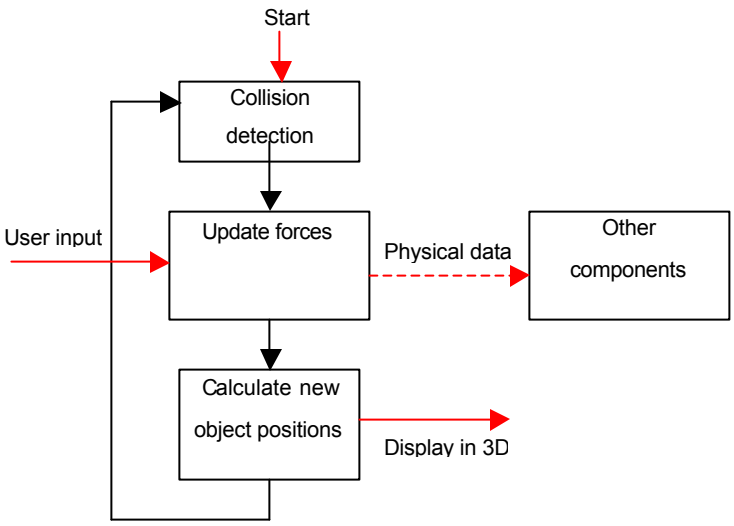

Figure 1.2 - *Physical simulation loop*

Two key issues for implementers of physics engines are dealing with collision detection and management of time steps. The collision detection process is responsible for rapidly flagging collisions between objects. A fast and robust method of achieving this is crucial and substantial efforts by the graphics research community have resulted in a variety of suitable algorithms (Hubbard, 1996; Cohen et al, 1995; O'Sullivan & Dingliana, 2001). The issue of time steps is also important to handle correctly, as a trade-off ensues between accuracy of simulation and processor load. The engine will step through the world in time steps as determined by the developer. The finer the time steps the more accurate the simulation, but also the more processor cycles required per step.

At this stage it should now be reasonably evident that the primary motive for the inclusion of a system like this in a game would be the improved animations and added realism. This is not the only reason though, as there are other benefits that physical simulation can provide for the developers, for instance:

?? In conjunction with the expected animation sequences resulting from the simulations, some additional behaviour that may never have been considered previous will emerge as the result of using dynamic simulations. Whereas all elements of an environment and its characters have to be considered when using key-framed animation, by using physical simulation the responsibility of considering every factor of the animation is taken away from the developer. Instead they will only have to consider the most basic physical attributes and any additional features of the animation will be produced as a result of the simulation (Watt & Policarpo, 2003).

?? As is the case with all developed components, reusability is a welcome feature. When using key-framed animation techniques it is rare to be able to recycle the animations from game to game as they invariably only cover the specifics of the application of the time and animators have to start from scratch each time. However, in an environment with its constituent objects being accurately modelled physically there is a good chance that they will be reusable in future developments.

From our point of view the use of rigid body simulation provides a wealth of accessible data pertaining to the physical processes underlying the simulation. Our contention is that this data can be used to compute sound as well as movement.

1.3 Research Goals

The primary goal of this project is to design and develop a system with which a games developer can easily add synthesised sound controlled by physical calculation to their game. Although physical simulation is concerned with modelling all forms of interactions between bodies, within the confines of this project, we are only concerned with direct body to body collisions and therefore will not consider interactions such as rolling and scraping. Our goal is to identify, implement and evaluate a technique which fits the following criteria:

- The synthesis method employed will need to be suitable for use with rigid body simulation.
- This method will not require any more data as input than that normally available from a modern physics engine.
- No pre-recorded samples should be necessary for the system to function correctly as this is precisely the type of method that the proposed system is expected to replace.
- The sounds produced by the system will need to sound enough like their real-world counterparts to constitute a plausible simulation.
- Neither the sound system nor the physically modelled objects should require any pre-processing before the system will function correctly.

Our aim is to determine to what extent the development of a system meeting the above criteria is possible and feasible.

1.4 Outline of Thesis

In this introduction we have described how the work presented in this thesis is an attempt at creating a generic sound synthesis engine for use in virtual environments. The approach takes advantage of the recent popularity of physical simulation in computer games to supply the necessary input for such an engine. The limitations of the current common practices of sound generation have been outlined and an alternative proposed. This was followed by an outline of a typical physical simulation system and a summary of our research goals.

Chapter 2 will describe both the history of sound synthesis, from both an analogue and digital perspective, and the current state of digital synthesis. Special attention will be given to modal synthesis as it was the synthesis method ultimately chosen for use in the implementation. The chapter concludes with the identification of a suitable synthesis technique for our purposes.

Chapter 3 details the other work that has been done in the area of digital audio synthesis, specifically work relevant to virtual environments and physical simulations. The usefulness to our own requirements of any of the discussed work will also be investigated.

Chapter 4 gives a comprehensive analysis of the various stages of the project's implementation, with particular attention given to the components of the constructed API and how the API should be integrated into other applications.

Chapter 5 deals with the procedures that the proposed system will be put through in order to test its functionality. Two test applications are described along with an analysis of the API from both developer and user perspectives.

Chapter 6 summarises what has been discussed throughout the thesis and presents the conclusions. Some proposals for future work in the areas of sound synthesis will also be considered.

Chapter 2

Sound Synthesis

In this chapter the various aspects of digital audio relevant to the project will be discussed. We begin with an overview of digital audio, where the specifics of its creation and properties are covered in detail. There follows an introduction into the area of sound synthesis containing a history of the use of synthesis techniques in audio applications over the last few decades. This section is mainly concerned with the musical applications of audio synthesis as this is where the bulk of the research and development was carried out in the past, but the principles are the same for our purposes. Following that is an analysis of some of the synthesis methods available to us for the implementation, including information as to their suitability for real-time collision sounds. We conclude with a detailed explanation of modal synthesis, which was the method finally chosen as the most suitable for the project, including derivations of the relevant formulae.

2.1 Principles of Audio

2.1.1 Audio basics

In this section we review the physical properties of sounds. More detailed treatments of this material can be found in any standard physics textbook, such as (Young & Freedman,

1996). In nature there are disturbances which travel from one region of space to another, and these are known as *waves*. Two main types of waves exist: *mechanical* and *electromagnetic*. Mechanical waves require the presence of some medium in order to travel from area to area, like air or metal. Electromagnetic waves do not require any form of medium and can travel through any space whether there is one present or not. We will only be dealing with mechanical waves here as that is the group to which sound waves belong. There are many types of mechanical waves that either occur in nature or are generated artificially, such as ripples in a pond or the motion of a taut string, but the most common and most important in our day-to-day lives are sound. Not only is sound used as the most common form of communication between people, but it also allows us to pick up additional information about our environments that is not conveyed through visual means, for instance the warning sound of an oncoming vehicle on the road.

Sound waves are best described as fluctuations in air pressure along the direction of propagation of the wave. These waves usually travel out in all directions from their source, but for the purposes of illustration, we concern ourselves only with an idealised case that propagates along a single vector. Although the diagram in figure 2.1(a) is the most physically accurate model of sound propagation, it is more convenient for us to represent waves visually by plotting the amplitude (defined later in this section) against time, as illustrated in figure 2.1(b). When an object is creating a sound, often as a result of an impact on its surface, it will begin to vibrate. These vibrations are transferred from the object into the air in the form of pressure fluctuations above and below that of atmospheric pressure. The diagram below illustrates how the high and low pressure areas, known as *compressions* and *rarefactions* respectively, correspond to the more common visual representation of a waveform. The eardrum, within the human ear, can detect these fluctuations in the air and interpret them to be what we define as sound.

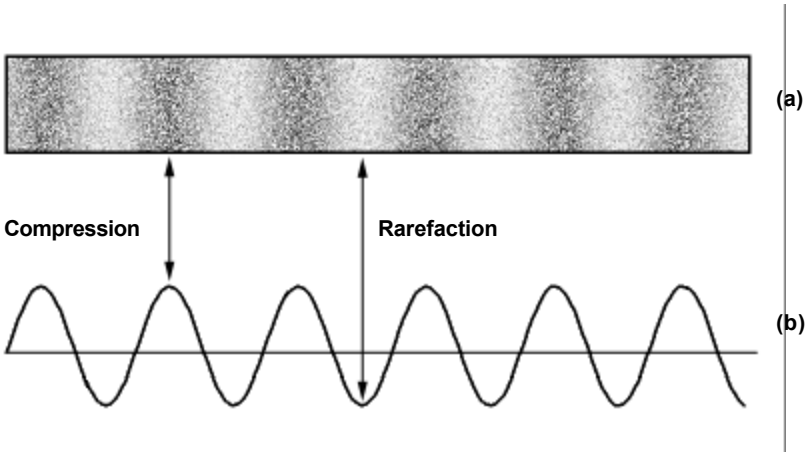

Figure 2.1 – *Example of both representations of waves*

In order to be able to understand and implement any synthesis techniques it is necessary to comprehend the fundamental properties of waves. The descriptions and formulae will assume the representations of waves are in the simpler continuous graph format (as shown in figure 2.2) and that the waves themselves are sinusoidal in form. Some of the more important properties include:

?? *Wavelength*

The wavelength of a signal is the distance from one crest on the wave to the next crest along the time axis (see figure 2.2). As wavelength is an expression of distance it is measured in metres.

?? *Frequency*

The frequency of a wave is a measure of how often the signal repeats per unit time. This property is also directly responsible for determining the pitch of the sound. It can be calculated by the ratio of the speed of the wave to its wavelength. Frequency is measured in units of Hertz (Hz), with a value of 1 Hz meaning the signal repeats once per second. The *period* of a wave is inversely proportional to its frequency and is an indication of how long it takes for one wavelength to pass a point in space. It is measured as the reciprocal of the frequency.

?? *Amplitude*

For the purposes of audio analysis, amplitude is a measure of the magnitude, or volume in the case of sound, of a signal. In reality the amplitude is a measure of the pressure changes in air, recorded as the difference from atmospheric pressure, in either the positive or negative directions. Amplitude is measured in metres.

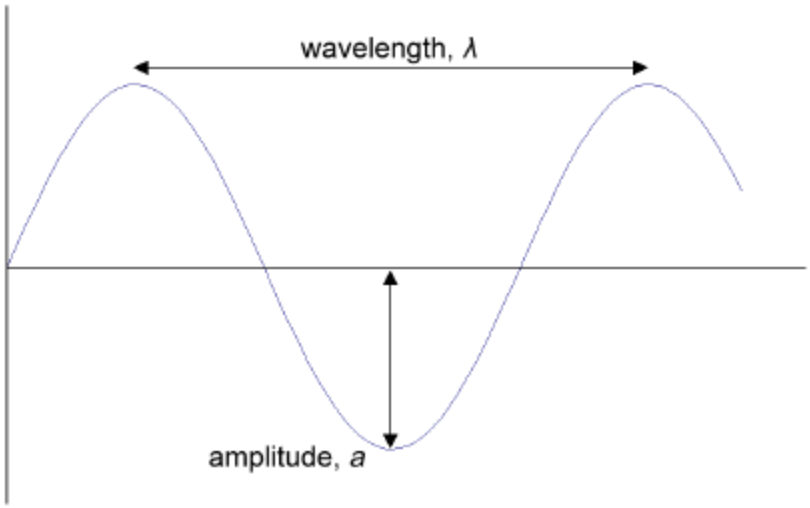

Figure 2.2 - *Illustration of values of wavelength and amplitude*

?? *Phase*

The phase of a periodic signal is an indication of which point along the waveform is the starting point. For a single waveform this has little significance other than to cause a positive or negative delay in playback, but when combining multiple waveforms together to form a single wave (a common operation when implementing synthesis techniques) the phase of waveforms relative to each other is vitally important. Phase data is also important in the production of short transient sounds, something which will be covered when synthesis methods are examined later. An example of how different phases can affect combined waves is shown in the diagram below (figure 2.3):

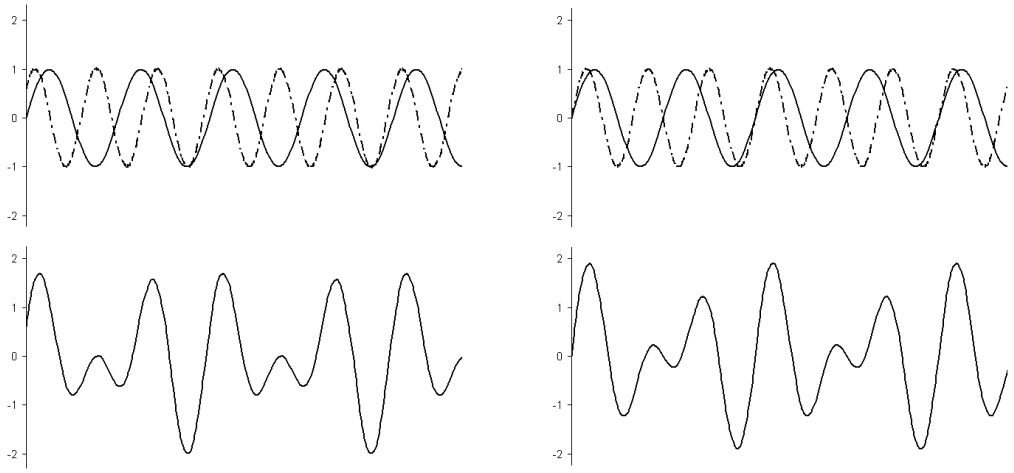

Figure 2.3 - *Example of how phase can affect wave addition*

?? *Bandwidth*

The difference between the highest and lowest frequencies of a transmission channel (the width of its allocated band of frequencies). The relevance of this term will be explained when sampling is discussed in the next section.

Waves can exist with a variety of compositions over the time they are active, but one of the more important types of waves for both the analysis and synthesis of sounds is that of the sine wave, or the similar cosine wave (which can be regarded as a sine wave out of phase). The sine wave can be represented mathematically by the following formula, which gives the instantaneous displacement y of a particle in the medium at position, x at time t:

$$y(x,t) \; ? \; A\sin(\; ? \; t \; ? \; kx) \qquad (2.1)$$

where A is the amplitude of the wave, ? is the angular frequency (equivalent to the normal frequency of the wave) and k is the wave number (a constant used to simplify the wave equation, generally equivalent to $2? / ?$). The sine wave is included here as it is a fundamental concept upon which many sound analysis and synthesis methods are based, especially Fourier theory and analysis.

2.1.2 Fourier Theory

Most sounds are not composed of single waveforms of a single frequency. In reality sounds are made up of juxtapositions of numerous waveforms. Luckily Fourier theory provides us with an important tool for the analysis of complex waveforms of this kind and ultimately leads us to a method whereby complex waveforms can be decomposed into simpler components. Fourier theory describes the concept that almost any signal, or wave, can be approximated by the sum of sine waves, each possessing a unique frequency. The more sinusoids included in the sum, the better the approximation. The mathematician Fourier proved that any continuous function could be produced as an infinite sum of sine and cosine waves. His results have far-reaching implications for the reproduction and synthesis of sound. A pure sine wave can be converted into sound by a loudspeaker and will be perceived to be a steady, pure tone of a single pitch. The sounds from orchestral instruments usually consist of a fundamental frequency component and a complement of harmonics, which can be considered to be a superposition of sine waves of a fundamental frequency f and integer multiples of that frequency. Hence by simultaneously playing each of these components together we can recreate the original sound (Zonst, 1995).

One of the important ideas for sound reproduction which arises from Fourier analysis is that it takes a high quality audio reproduction system to reproduce percussive sounds or sounds with fast transients (the initial moments of the sound). The sustained sound of a trombone can be reproduced with a limited range of frequencies because most of the sound energy is in the first few harmonics of the fundamental pitch. But if we are going to synthesize the sharp attack of a cymbal, we need a broad range of high frequencies to produce the rapid change. We can visualize the task by adding up a group of sine waves to produce a sharp pulse and see that we need large amplitudes of waves with very short rise times (high frequencies) to produce the sharp attack of the cymbal. This insight from Fourier analysis can be generalized to say that any sound with a sharp attack, or a sharp pulse, or rapid changes in the waveform like a square wave will have a lot of high

frequency content. Many operations on digital audio, such as filtering, are accomplished through the use of an implementation of the Fourier theory known as Fast Fourier Transforms. Strictly speaking, this form of analysis can only be applied to periodic waves, i.e. signals that repeat indefinitely. For details of the mathematics behind Fast Fourier Transforms the reader is referred to digital signal processing (DSP) literature, e.g. (McClellan et al, 1998).

2.1.3 Digital audio

It is important at this stage to make the distinction between two different categories of audio signals: analogue and digital. An analogue signal is one which is continuous in amplitude and time. The nature of these types of signals causes problems when trying to record them on modern digital systems, due to the infinite amount of data that would need to be stored to represent them. No matter how small a measurement of the original signal was taken there would always be one smaller again. This was not a problem for older recording mechanisms, such as records and magnetic tapes as they used analogue storage systems, but the current era of storage media uses digital means and hence requires the signals to be converted. This is done through the processes of *sampling* and *quantization*.

Sampling involves taking a measurement of the amplitude of a signal at regular intervals. A *sampling rate*, which determines how many times a second the amplitude will be recorded, must be chosen that will preserve at least the full range of audible frequencies, assuming that near perfect quality is desired. A fundamental law of digital signal processing states that if an analogue signal has a maximum frequency of B Hz, then the signal can be periodically sampled at a rate of $2B$ Hz, and reconstructed accurately from the samples. This optimal sampling rate is known as the *Nyquist frequency*. Sampling at anything above the Nyquist frequency will only generate extraneous data, but will not improve the quality of the recorded sound. Any frequency that exists in the wave to be sampled that is above the decided upper frequency will still be sampled along with the

other components, but will be stored incorrectly (possibly interfering with other frequencies) due to not enough data being taken. Sampling at values below the Nyquist frequency will result in loss of data. The diagram below (figure 2.4) illustrates the adverse effects of using too small a sampling rate. The original wave on the left bears a resemblance to a sine wave, but on closer examination one can see that there are many smaller variations along that path. By using such a low sampling rate the resulting digital signal will only have captured the larger changes in the wave and ignored the more subtle changes.

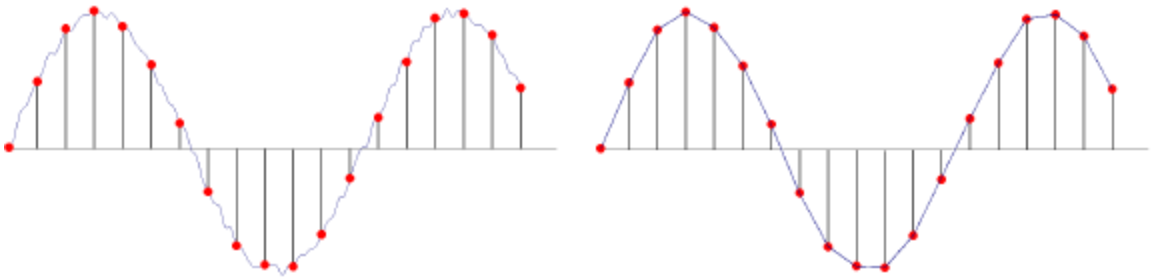

Figure 2.4 - *Left: original wave, showing sampling points. Right: resampled wave.*

The second factor in the process of analogue to digital conversion is quantization. This is similar to the sampling just described, but instead of dealing with the frequencies of the signal, quantization deals with the conversion of the amplitudes of the signal. Also, like with sampling, a value is chosen, called the *quantum value*, which determines the minimum step between two successive amplitude values. Each time the wave is sampled according to the sampling rate a value for the amplitude is taken, with the value being rounded off to the nearest multiple of the quantum value. In the same way that sampling reduces the continuous time values into discrete values for digital recording, so too does quantization follow the same procedure for recording amplitude. Unlike sampling however, there is no formula for determining the optimal quantum value; instead it is left up to the developer to decide what is sufficient for the required application. Naturally a smaller value will result in a higher quality, but will also require more data storage. CD-

quality sound, which is considered perfect quality to the human ear, is sampled at 44 kHz and quantized to 16 bits.

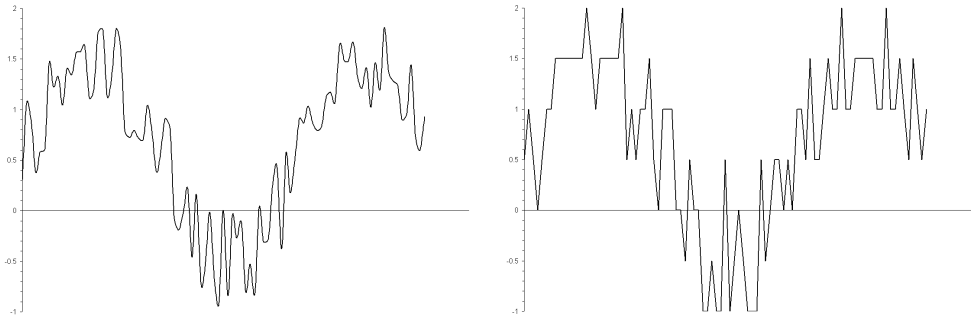

Figure 2.5 - *Left: original wave. Right: sampled wave, showing quantization error.*

2.2 Origins of Sound Synthesis

The primary focus for research into sound synthesis to date has been in the area of music, with applications to sound effects and other similar ideas being a more recent development. The work has involved trying not only to re-create the sounds generated by real instruments, but also to create new sounds that had not been heard before. As such, many of the methods developed are not appropriate to the fulfilment of this project, as they are geared almost exclusively towards the production of digital musical instruments. However, it is still informative to examine the evolution of these methods throughout the last century.

It has already been demonstrated in previous sections that there is a significant difference between analogue and digital audio signals. Forms of analogue synthesis predated the introduction of digital computers, with the first synthesiser having been invented in 1791 by Wolfgang von Kempelen (Dutoit, 1997). His device was a mechanical model of the human vocal system, composed of a main bellows, reed and India-rubber cup to simulate the lungs, vocal chords and mouth respectively. It was capable of mimicking about twenty different speech sounds. The first of the modern musical synthesisers was created

at the turn of the century in 1906 by Thaddeus Cahill (Iazzetta & Kon, 1999). It was called the Telharmonium, or Dynamophone, and consisted of shafts and inductors that produced alternating currents of different audio frequencies. This machine was actually a modified electrical dynamo and, using a number of specially geared shafts and inductors, it could produce alternating currents of different audio frequencies. These audio signals passed via a keyboard and associated bank of controls to a series of telephone receivers fitted with special acoustic horns. The whole machine had a weight of about 200 tons, was about 60 feet in length and cost in excess of $200,000. Unfortunately the Telharmonium proved unpopular due to its excessive weight and size and the fact that it caused serious interference to telephones and thus it was shunned as being impractical. The basic principles underlying the structure of this device were later used to good effect in the 1950's and 1960's in the form of the Hammond organ.

The first digital synthesis experiments were conducted in 1957 by Max Matthews, et al. in Bell Telephone Labs (Roads, 1996). In these early experiments, Mathews and his colleagues proved that a computer could synthesise sounds according to any pitch scale or waveform, including time-varying frequency and amplitude envelopes. These experiments were a long way from what we know today as digital sounds, and the idea of real-time synthesis was still a good way off. In fact the calculations for the sound were written directly in terms of machine instruction language for powerful computers at IBM in New York and then transferred back to the Bell Labs in New Jersey, where the resulting computations were then played on a much less powerful machine that had audio capabilities. In the years that followed, Matthews was involved in creating the musical composition languages, titled Music, from version I upwards. The original Music program, written solely by Matthews in 1957, could generate only a single waveform type, only allowing the user control over the pitch and duration of each tone. Music II was written a year later for a more powerful IBM computer. This version of the program increased the number of possible waveforms to 16 and also contained four independent channels of sound. This was a big step from the previous version, but in 1960 a new development appeared in the form of the *unit generator*.

2.2.1 Unit generators

Unit generators are basically single components which can be combined together in various ways to produce synthesis instruments, also known as patches. These patches can then generate complex sound signals. The unit generators come in two types: signal generators and signal modifiers. The signal generators are what create the generic sounds in the beginning of the synthesis process, with the signal modifiers altering the signals in some way for output. Some examples of unit generators are:

?? *Oscillator*

An oscillator is a signal generator which produces a single tone at a particular frequency and amplitude, and it is the first element to be triggered in a synthesis process. In some instances a pre-recorded sample or a live input may take the place of the oscillator, depending on what the desired output is.

?? *Filter*

Filter is a term which generally encompasses any component which performs an operation on an input signal and returns the output. Common examples of filters include *low pass*, *high pass*, *band pass* and *band reject*, all of which affect the signals in unique ways. These fall into the category of signal modifiers, since they alter the signals they take as input.

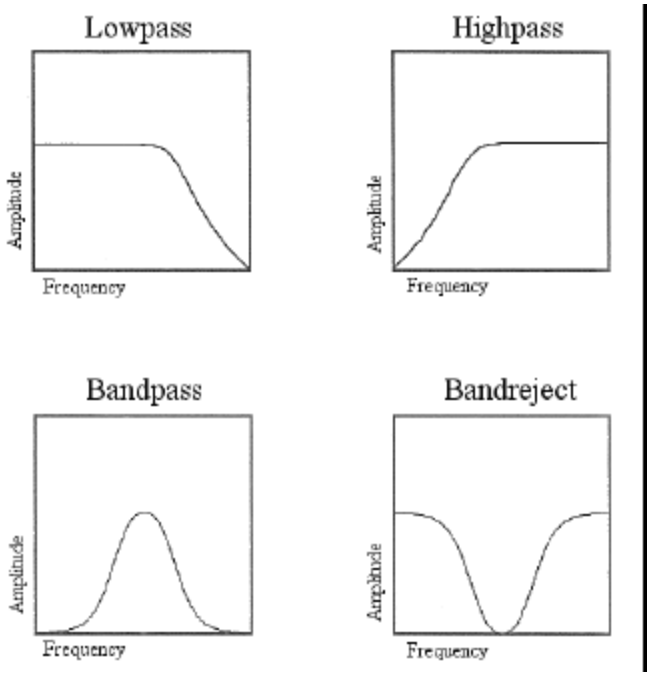

Figure 2.6 - *Examples of filters*

Figure 2.6 illustrates how the above named four filters would affect an input signal. Each of these filters will alter the input signal by reducing the amplitudes of the components with frequencies that fall into the range the filter is designed to block. For instance, the low pass filter will remove all frequencies that are above a certain cut-off value and the band pass will only allow frequencies close to a specific value to remain.

?? *Envelope*

An envelope is another form of signal modifier which controls another property of a wave, often the amplitude (volume), over time. This can be used to create effects like fading in and fading out or Doppler shift, which is the change up or down in pitch as a sound source moves towards or away from the listener.

With the use of unit generators a completely modular system can be constructed. It is no longer necessary to create systems from the ground up; instead new ones can be built from the components of existing systems by adding in only the new features.

Originally these components were hardware elements whose precise design was made possible by the careful application of Fourier theory. Increasingly researchers realised that these components could instead be implemented with software. This idea was used by Max Matthews and John Miller that same year to develop Music III. Unlike the two prior versions, this one allowed the user to generate a large variety of synthesis techniques, thanks to the use of the unit generators. Music IV and V were developed a few years later, with version V being ported to a multitude of platforms, serving as a good introduction to digital sound synthesis for a wide audience. Since the advent of the unit generators, and in particular their use in Music III, IV, and V, a vast array of synthesis programs and compositional languages has appeared. The most popular today is CSound (Boulanger, 2000). This package is actually a synthesis scripting language, rather than just a music maker. Given the vast array of sound generation methods available to us, not only in the above mentioned packages, but in computer systems in general, it is advantageous to look at a number of the more significant ones, while examining their applicability to this project.

2.3 Synthesis methods

When the first PCs began generating sound, they were a far cry from what we have come to expect from them today. In the early days the only sound generator available to the masses was a simple tone generator attached to a speaker, now referred to as the PC speaker. This device could only generate tones somewhere in the range of 40 Hz to above 30 kHz and could only play a single tone at any particular time. While this served the requirements of a PC in its day, it was a decidedly primitive device when attempting to create a soundtrack to a computer game and the sound effects and music in those days were more of a hindrance than a welcome addition. Only having control over the length and frequency of the tone did not provide enough of a range for games developers. This was all to change with the introduction of the SoundBlaster sound-card from Creative Labs in 1989. This brought forward the audio capabilities of the home PC by a staggering amount (Demara & Wilson, 2002). It was now not only possible to play many

sounds simultaneously, but the programmer was no longer limited to using simple "beeps"; they could now record live sounds and play them back on request. It was still possible to synthesise the same simple sounds mathematically, but the composer was now given much more control over the parameters of the sounds.

2.3.1 Sampling

With the introduction of these SoundBlaster cards and their rivals, sampled sounds became the de facto standard for audio in computer multimedia and entertainment. Although this would not seem to be a form of synthesis, as explained previously in the last section, the data that is stored on the computer is in fact only a representation of the original sound rather than an exact duplicate. The majority of sounds recorded in the early days of sound cards were stored using pulse code modulation (PCM), one of the now numerous encoding algorithms available to audio technicians. While this form of sound generation is often referred to as sampling, nowadays it is more common to refer to it as *wavetable synthesis*. Wavetable synthesis, in fact refers to any system by which the numeric data of a sound is stored in memory or in a file and is played back when required. Whether the data was generated through sampling an analogue signal or the results of a mathematical equation (like the sine wave function in section 2.1.1) is irrelevant. The main feature of wavetable synthesis is that the complete sound exists in its entirety before it is ever played, quite the opposite of the other synthesis methods that will be discussed later. Using this method the developers would have a pre-recorded sound for each sound-making action in the game stored on the disk. When a particular event takes place on screen the relevant sound would be loaded into memory from the disk and played back to the user. This is where some of the limitations of wavetable synthesis become apparent. When playing the recorded sounds back, there are a limited number of methods available to the developer to vary the sounds, restricting the developer to functions like changing the pitch or applying a new volume envelope. Changing the volume can be an effective method when trying to indicate the force of a collision, as invariably the greater the force of impact, the louder the sound. However,

there are more factors relevant to louder sounds than simple increases in volume; most sound sources exhibit spectral variations as a function of loudness, e.g. there is often more high frequency energy in loud sounds than in soft sounds [REF].

It was already discussed in section 1.1 that with the introduction of physical simulation into mainstream development, the repetitive nature of sample sounds was made even more obvious, with actions that looked noticeably different on screen providing the same sound effect. Before showing how some of the more mathematically complex synthesis methods can go some way to solving this problem, it is worth mentioning *multisampling*. This approach attempts to solve some of the limitations of wavetable synthesis by storing multiple recordings of sounds at different pitches, and switching or interpolating between these during resynthesis. It is also possible to store separate samples for loud and quiet sounds, eliminating the problem described above. This comes with its own set of problems though. One major issue is the additional storage space required to store all the new samples. Whereas a traditional game using pre-recorded sounds might have used about 100 or 150 sounds, with multisampling this number could be multiplied by a factor of as much as ten. In other words, what would have taken 20 MB of space, could now take as much as 200 MB. A follow-on from this which bears a passing resemblance to that of sampling is how many samples are required for each sound. The more samples that are stored for each possible sound (both for varying pitch and for varying volume) the greater the realism will be. Once again though, the increases in storage space put logical limits on this. It should be clear now that no direct variation on sampling is going to be able to fully solve the problems of realism and repetition, so the remainder of this section will deal with a number of synthesis methods that propose reasonable alternatives to sampling.

2.3.2 Additive synthesis

The most fundamental method of sound synthesis is additive synthesis. Additive synthesis is based on the principle that all sounds, even complicated ones, can be

analysed using Fourier analysis and described as a series of sine waves at different frequencies and amplitudes. In additive synthesis, a large number of sine waves are combined to produce a complex waveform. Dynamic changes in the waveform are created by varying the relative amplitudes of as many as several dozen of these sine waves. Additive synthesis has been around since the early days of sound synthesis, predating even the digital era, and is in fact a direct equivalent to how the previously mentioned Telharmonium and Hammond organ operated. Implementing the basic principles of Fourier theory, in that it is theoretically possible to approximate any complex waveform as the sum of elementary waveforms, was what gave this type of synthesis its versatility. This method however is much more suited to the synthesis of traditional musical instruments, e.g. violin, piano, where all the notes are harmonic, i.e. have frequencies which are integer multiples of the fundamental frequency, rather than percussion or sound effects, which have somewhat more irregular frequencies. Figure 2.7 shows two waveforms, sin(x) and sin(y), along with the resultant waveform from their addition.

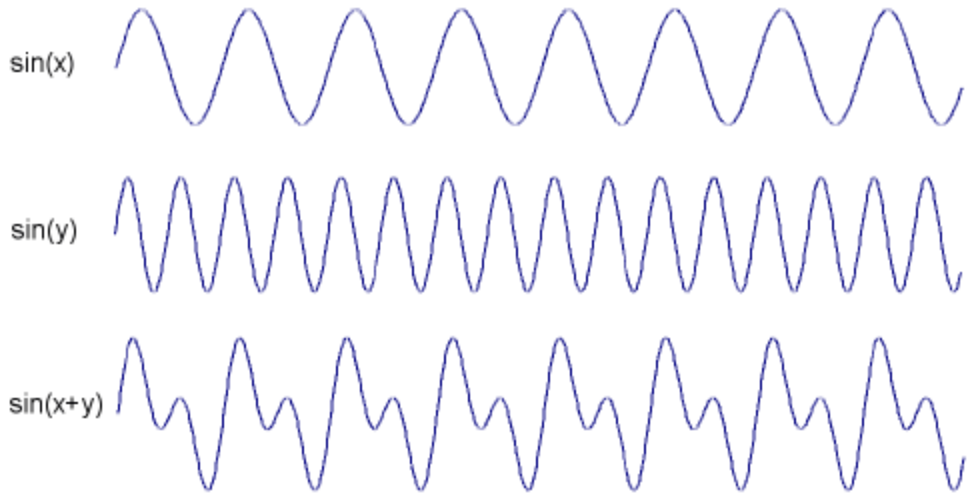

Figure 2.7 - *Example of additive synthesis*

2.3.3 Subtractive Synthesis

Subtractive synthesis is in essence the reverse of additive synthesis. A sample or a generated waveform which is rich in harmonics can be fed into a set of filters so that some harmonics can be reduced in amplitude or completely removed from the spectrum. For example, if a generated waveform has higher harmonics than are required, it can be passed through a low pass filter so that they can be removed or attenuated, leaving the lower harmonics unaltered. This form of synthesis has its uses, but they do not apply in this instance. In order to successfully implement a subtractive synthesis technique, an original sound is required and this is quite the opposite of what we hope to achieve. Subtractive synthesis is more suited to making minor modifications and modulations to existing sounds than to creating new ones.

2.3.4 Modal Synthesis

Modal synthesis (van den Doel et al, 2001; van den Doel et al, 2003; Cook, 1996) is something of an extension of additive synthesis in that the final sound is created through the addition of other simpler waves, but the methods by which these simpler waves are generated is more complex. Modal synthesis aims to provide exact simulation of the audio properties of specific real-world objects and does so by representing these objects as collections of many smaller vibrating objects, or substructures. The fundamental theory behind it is that each of these substructures has its own set of natural modes of vibration, which are exhibited whenever the structure becomes excited; in the specific case of this project this would be due to collisions between objects, but it can also occur due to sliding, scraping, pressures, etc. In modal synthesis these mini-objects are modelled mathematically as a set of modal data, consisting of the frequencies and damping coefficients (rates of fall-off of amplitude over time). The waves generated from this modal data are then added together to give the fully synthesised sound effect. The sources of this modal data will be discussed in the next chapter when some of the other research into the applications of modal synthesis is examined. Naturally there is

much more to this synthesis algorithm than described here, but the specifics of modal synthesis will be dealt with later in detail in section 2.4, including derivations for the mathematical representations of the algorithms.

2.3.5 Granular Synthesis

Granular synthesis is a technique which involves a complete sound from thousands of sound *grains*. Each of these grains lasts only for a brief period, usually less than 100 milliseconds; such short durations being only just beyond the threshold of human hearing. This procedure was first proposed by (Gabor, 1946) and later developed further by researchers like (Bastians, 1980; Hoskinson & Pai, 2001). By dividing up the sounds into such small components, it provides the composer with complete control on a grain-by-grain basis. The source of these grains can vary and may come from a wavetable, FM synthesis or sampled sound. Unfortunately, while granular synthesis may be capable of producing some very convincing results, its dependency on existing samples (however small they may be) and lack of physical representation compared to modal synthesis makes it unsuitable for our goals. The computation time required for this form of synthesis is another factor which makes it unsuitable for us. Although it would be possible to generate the data for a single sound in real time, extending that to cover a variety of simultaneous sounds would be unfeasible with current technologies.

2.3.6 Conclusion

Ultimately, modal synthesis was chosen from the above methods and others as the synthesis algorithm most suitable to the needs of the project. Since it deals specifically with the sounds generated from collisions (even if it was designed for musical instruments) it seems to be the ideal choice for a project that deals only with collision sounds. If the engine was ever expanded beyond those criteria then the limitations of using modal synthesis might become apparent, but for the time being it is the best choice. With that in mind, there follows a detailed explanation of the workings of modal

synthesis, including a derivation of the fundamental formulae. More detailed treatments of these synthesis methods can be found in texts such as (Roads, 1996).

2.4 Modal synthesis

We now present a more detailed description of the modal synthesis process. Modal synthesis represents objects as collections of many smaller vibrating objects, or substructures. The fundamental theory behind this is as follows. Any object, when struck, vibrates rapidly. These vibrations cause the displacements in air pressure that we perceive as sound. In order to model such a vibration, with a view to synthesising the resultant sound, we need to know three things:

a) frequency of vibration
b) amplitude of vibration
c) damping co-efficient

In theory, this should be simple to model, but the difficultly lies in the fact that objects do not vibrate at single frequencies, but rather their physical structure causes their surfaces to vibrate at a myriad of different frequencies. Modal synthesis proceeds by identifying these characteristic modes of vibration for a particular object and modelling the object in terms of these modes, so a model of an object might consist of an array of pairs (F_i, D_i), where F_i is a frequency and D_i is the corresponding damping coefficient.

This still leaves us with the question of determining an appropriate mathematical model for how an individual mode should be sounded over time; in particular how the damping should be handled. A good physical system which is used as a basis for this is the mass-spring system (illustrated in the diagram below, figure 2.8) Considering that in acoustics the modes are produced by vibrating substructures of the object, the use of masses and springs to model them is a logical correlation. If, in one of these systems, the mass is pulled (or pushed) in a particular direction, then the spring will attempt to restore the system to its rest state, initially by applying a force in the opposite direction to the

original movement. While the system is attempting to return to rest, it will behave like a simple harmonic oscillator. The important factor here is the damping and this is also the reason why we can not just generate a simple sine wave at a specified frequency and amplitude. The damping determines the duration of the oscillation, which in sound terms is an indication of how long it will be audible before fading to zero. The higher the damping rate, the shorter the sound.

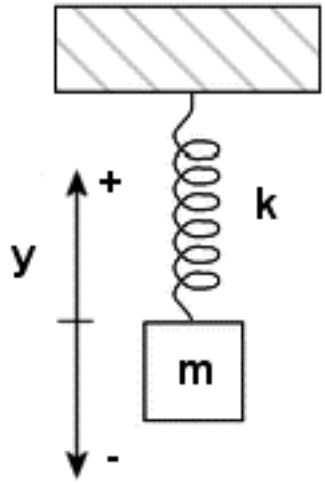

Figure 2.8 - *Mass-spring system*

Physical symbols used in this diagram are mass m, displacement from rest position y and stiffness of spring k. The stiffness of the spring is defined as the force required to displace the spring from its rest position. As with any physical system, energy losses occur and we represent this by a damping factor, r. Newton's second law of motion ($F = ma$) can now be applied in order to analyse the forces and their effects in the system, resulting in the following equation:

$$-ky - mg - rv = ma \qquad \text{(Force = mass x acceleration)}$$

Here the new variables for gravity g, velocity v, and acceleration a are also introduced. k is multiplied by the displacement because the larger the displacement of the mass, the stronger the resultant force on the spring. r is multiplied by the velocity because the

faster the mass is oscillating, the more the damping forces come into play. Compared to the magnitude of the force exerted by the deformed spring, the force exerted by gravity is negligible, so we can simplify the equation by removing this operand. This results in the following formula:

$$-ky - rv = ma \qquad (2.2)$$

In order to get a solvable equation, it becomes necessary to make some substitutions. Knowing that velocity v is the rate of change of position y with respect to time t, and acceleration is the rate of change of velocity with time the following formulae can be achieved:

$$-ky - r\frac{dy}{dt} - m\frac{d^2y}{dt^2} \qquad (2.3) \qquad \text{or}$$

$$\frac{d^2y}{dt^2} + \left(\frac{r}{m}\right)\left(\frac{dy}{dt}\right) + \left(\frac{k}{m}\right)y = 0 \qquad (2.4)$$

We need to solve for y. Since we are going to be dealing with the discrete case there is no need to try and solve this differential equation. Instead we can make some approximations, namely that:

$$\frac{dy}{dt} - \frac{y(n) - y(n-1)}{T} \qquad (2.5)$$

$y(n) - y(n-1)$ represents the change in position from sample $n-1$ to sample n and T represents the length of the sampling interval. Therefore this represents an appropriate approximation of velocity. Similarly:

$$\frac{d^2y}{dt^2} - \frac{dv}{dt} = \frac{\left(\frac{y(n) - y(n-1)}{T}\right) - \left(\frac{y(n-1) - y(n-2)}{T}\right)}{T}$$

$$= \frac{y(n) - 2y(n-1) + y(n-2)}{T^2} \qquad (2.6)$$

If we now put equation 2.6 and equation 2.5 back into equation 2.4 we get:

$$\frac{y(n) - 2y(n-1) + y(n-2)}{T^2} + \left(\frac{r}{m}\right)\frac{y(n) - y(n-1)}{T} + \left(\frac{k}{m}\right)y(n) = 0$$

$$y(n) = \frac{y(n-1)(2m - Tr) + Y(n-2)(m)}{m + Tr + T^2k} \qquad (2.7)$$

Effectively, we now have a way of calculating, for a given mass-spring system, the position of the mass at any moment in time. The motion is simple harmonic in nature which is a motion that maintains constant frequency and decays in amplitude according to the damping co-efficient. This is also an ideal method of modelling the decay in amplitude of a vibrating mode over time. This equation can be implemented through use of digital electronics like resonant filters, or in our case simplified algorithms for a computer program that simulates this.

However, knowing how to solve a single mass/spring system or having a single modal filter is only a small part of being able to physically model an object. The required modal data must be obtained from somewhere. Some of this data is available in the public domain, or else it can be obtained experimentally through the use of existing equipment in the industry, such as the Active Measurement Facility (ACME) at the University of British Columbia, which has the capability to automatically acquire sound measurements by moving a sound effector around the surface of a test object by the use of a robot arm (Pai et al 1998, 2001). A more direct physical modelling approach would be to form a type of mesh to represent the object in all dimensions, and to do this it will be necessary to have multiple masses connected together by multiple springs, or its equivalent using digital electronics (O'Brien et al. 2002). This not only makes the mathematical calculations more complicated, but it also increases the processing power and time that

will be required for a real-time system to be able to keep up with the graphical simulation. Once the waveforms have been calculated for each sub-component, they are combined using basic sinusoidal waveform addition.

This chapter illustrated the fundamental aspects of computer audio, from the basics of wave theory to the intricacies of synthesis methods. The next chapter will be used to expand on these theories by showing some of the academic and practical applications that have been attained using the methods previously described, while also demonstrating the different approaches available for acquiring the modal data.

Chapter 3
Sound Synthesis in Computer Games

In this chapter we will continue what was discussed in previous sections by looking at some specific applications of modal synthesis in modern virtual environments. While modal synthesis provides many benefits in terms of convenience and low computation time, the main stumbling block to its application is the difficulty involved in determining modal data for specific objects. A number of different methods have been suggested to provide the developer with the necessary modal data for the objects they wish to simulate. Each of these uses a different approach to the problem and we can provide a rough taxonomy which divides them into a number of broad categories as follows:

- ?? Finite elements
- ?? Measurements
- ?? Deformable models
- ?? Mathematical derivation

The remainder of this chapter will discuss the various methods of implementing modal synthesis under the above headings, followed by a conclusive analysis of how they apply to this project.

3.1 Finite element models

A finite element method divides the shape of the object into fixed elements or tetrahedral shapes. Just as a mesh of triangles can be used to approximate any surface, so too can tetrahedral shapes be used to approximate an arbitrary volume. For each of these tetrahedra, or elements, the material is represented by a function with a finite number of parameters. The function itself can further be divided into more basic functions, with each one of them representing one of the nodes, or vertices, on the boundary of the element. Elements which are adjacent to each other will naturally have nodes in common, so this reduces the overall size and complexity of the data-set considerably. The complete finite element mesh is then represented by a composite function composed of all the base functions of the nodes and elements. A physical system that has been discretised using a finite element, finite differencing, or similar method can be expressed in the following general form:

$$K(d) + C(d, \dot{d}) + M(\ddot{d}) = f \qquad (3.1)$$

where d is the vector of node displacements, K and C are nonlinear functions that respectively determine the internal forces due to node displacements and node velocities, M maps node accelerations to node momentum and f represents any other forces. In one sense a finite element system can be regarded as a highly complex mass-spring system where the nodes are the masses and the connections between the nodes are the springs. In a similar way, forces that act on the model will propagate mode displacement throughout the system, resulting in oscillations of the surface nodes. These surface node oscillations correspond to the modes of vibration of the object. Therefore, if we model an object in this manner, we end up with a mass-spring system that not only models the

amplitude decay correctly for each mode, but also computes the actual mode frequencies as well. Methods for solving equation 3.1 above in order to achieve this are well described in engineering literature, such as (Buchanan 1994).

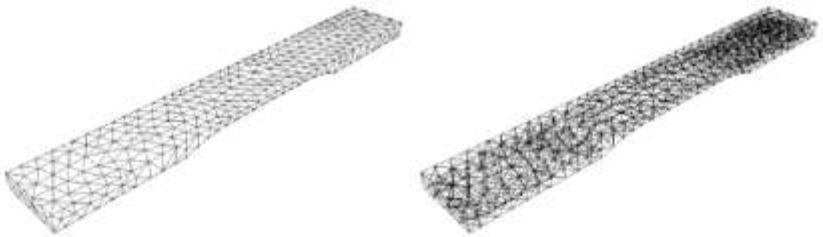

Figure 3.1 - *Finite element mesh showing both external forces (left) and internal structure (right)*

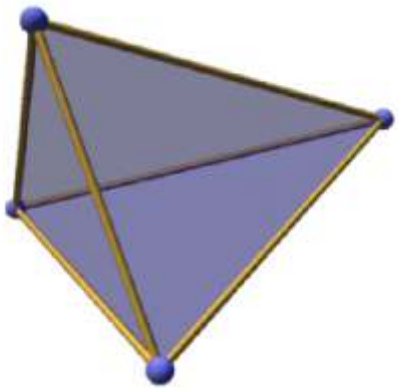

Figure 3.2 - *A tetrahedral element, showing the four nodes*

In (O'Brien et al. 2002) the authors describe their approach for creating a soundtrack to correspond to a rigid body simulation. Their approach entails pre-computing the shape and frequencies of an object's deformation modes and then interactively synthesising the audio directly from the force data generated by a standard rigid-body simulation. There are three main stages to the process described in this paper:

1) Convert polygonal model to finite element mesh.
2) Extract modal data from finite element mesh.
3) Use modal synthesis to generate sounds in real time.

We now look at each of these three stages in more detail. For the purposes of the final simulation, a polygonal model will have been created to represent each object, both visually and in the rigid-body simulation. This model is only a model of the objects surface, so since the aim in the second stage is to analyse how impacts on the object would inflict deformations through it (not just on the surface) then a system is required which can represent the object in true three dimensional form. The authors use the finite element method for achieving this. The amount of time it takes for decomposition of a polygonal model and the corresponding material into a finite-element model varies greatly depending on the complexity of the original model and thus the number of modes it has. For the simplest of objects, such as a box, this can be computed quickly, usually in a few seconds. For the more detailed models like a game character the time can take anything from an hour upwards. For example, the authors report that a polygonal mesh of a cuboid shape with 361 nodes took only 30 seconds to compute. In comparison, a more complex object in the form of a detailed rabbit model took almost 5 hours to compute.

Once the environment has been discretised using the finite-element the next step is to perform modal analysis on the system to extract the modal data for use in the final simulation. In its most general form the functions that represent the finite-element model are non-linear, and hence so is the overall function. However, since the displacements experienced by the elements of the model are small the equation can be linearised. Much of the modal analysis takes the form of complex mathematical operations on matrices and is beyond the scope of the aims of this section of the thesis. Ultimately, what begins as a non-linear matrix representation of the model is reduced to a relatively simple equation which can provide the frequency of each mode and the decay rate of the same mode. For further details on these methods the reader is referred to [REF].

Initial impressions would suggest that the work done in this paper is the closest to meeting the requirements of our project. To begin with, the system uses rigid bodies in the real-time simulation as opposed to the deformable models which some of the other papers we will look at use. On top of that, although a custom physics engine was used for the research described, the authors do point out that they found that useful results can be generated using more generic rigid-body simulators which bodes well for our proposed system being independent of any specific physics engine. This is due to the fact that once the modal data has been calculated using the finite-element model, there is no longer any need to model the minute deformations that results from sound making collisions.

However, there is one area in which the methods here fail to meet the criteria put forward in our project goals and that is the amount of pre-processing involved. In our original specifications, we detailed how neither the sound system nor the physically modelled objects should require any pre-processing before the system will function correctly. Even though is takes only thirty seconds or a minute to process one of the simpler objects, when you take into account that there could be anything up to 100 or more objects in a modern computer game and the fact that most of them will be more complex that simple boxes and spheres, then it can be seen just how much time the pre-processing would actually take for a fully realised system.

3.2 Measurement

(van den Doel et al. 2001) describes the research and development of a software system called FoleyAutomatic, composed of a dynamics simulator, a graphics renderer, and an audio modeller to create interactive simulations with high-quality synthetic sound. Unlike some of the other papers covered here and also our own project, the research detailed here is not limited to mere impact interactions, but also constructs a model to cover rolling and sliding as well. The paper also covers the extension of modal synthesis to incorporate multiple simultaneous continuous interactions at different locations with

realistic timbre shifts depending on the contact point. These same procedures are also covered in another paper by the same authors (van den Doel & Pai 2003).

The modal synthesis methods described in this paper take a similar approach to those previously described in section 2.4 of this thesis, although the mathematical equations are represented differently. The modal model $? ? \{f, d, A\}$ consists of a vector f of length N whose components are the modal frequencies, a vector d of length N whose components are the decay rates, and an N x K matrix A, whose elements a_{nk} are the gains for each mode at different locations on the surface of the object. The modelled response for an impulse at location k is given by:

$$y_k(t) ? \sum_{n?1}^{N} a_{nk} e^{? d_n t} \sin(2? f_n t) \quad \text{for } t ? 0 \qquad (3.2)$$

The equation above also models a form of simple harmonic motion similar to the one we derived in section 2.4. For modal synthesis the frequencies and dampings (decay rates) of the oscillators are determined by the object's geometry and its material properties, such as elasticity and density. Also, the gains of the modes are related to the mode shapes and are dependent on the contact location on the objects.

A somewhat mixed direction is taken when determining how to obtain the modal data for the synthesis. Where other authors have tended to choose one method for obtaining the data, here the methods chosen vary depending on the object in question. For simple objects the authors choose to either derive the modal model parameters from first principles as described in (van den Doel & Pai 1996) or to use a modal model editor to create the modal data by trial and error. However, for the more complex objects in an environment, or to simply get a more realistic synthetic sound, they obtain them by fitting the parameters to recorded sounds of real objects, using the Active Measurement Facility (ACME) (Pai et al. 1998).

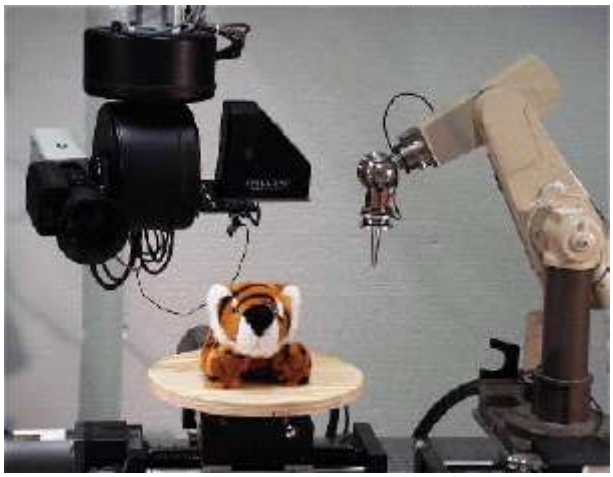

Figure 3.3 - *ACME facility with test subject*

Just as when modelling an approximation of a real world object for visual representation, the authors also create a representation here of the object with respect to its sonic properties. Each point on the surface of the object that was sampled using the ACME device is then mapped onto the *sound-space* of the computer model, which they define as the space spanned by the gains of the modes. This creates a sparse lattice on the surface of the object with each lattice node having exact parameters for the correct synthesis of sounds resulting from impacts at any of those points. The density of this lattice has a strong influence on how accurately the synthesised sounds will reflect those of the real world. At intermediate points an interpolation scheme is used to evaluate the nearest parameter values.

Since our project is concerned solely with the production of sounds resulting from collisions, the systems for rolling and scraping described in this paper will not be discussed here. As a result of the broader scope of their efforts, the authors chose to use a different physical modelling representation than the one used in this project. Due to the discontinuity errors that would result from using polygonal models in rolling and sliding calculations, smooth surface models were instead implemented using piecewise parametric surfaces and loop subdivision surfaces allowing for more accurate simulations.

This paper notes a number of positive outcomes of the research. One of the more interesting results that the authors comment on is how, through experimentation with various different models, they found that the exact details of the shape of objects are relatively unimportant. Also important is the relatively high number of modes that their system was able to compute in real-time. On a 900 MHz Pentium III it was found that about 800 modes could be synthesised at a rate of 44100 Hz. They point out that this is good enough for approximately ten reasonably complex objects while still giving processor time to graphics and other simulations components. However, this is probably using more modes than is necessary (eighty modes per object), when something between ten and fifteen would be more appropriate and still give a convincing sound. Another relevant point they make is that for loud noises it is sufficient to have only four modal frequencies playing to give a convincing sound, due to how these more prominent frequencies would drown out the lesser ones if they were present. The obvious problem with this approach is that its effective development relies on the ability to construct this lattice of frequency / damping pairs across the surface of the object.

3.3 Deformable models

While the previous papers discussed have all focused on rigid body simulation, there are a number of other approaches that can be taken to effectively simulate objects and synthesize sounds. One of these different directions is to use non-rigid, i.e. deformable, models. This is the method that the authors chose to implement in (O'Brien et al. 2001).

Other work by these authors (O'Brien et al. 2002), already discussed, used finite element methods to pre-compute frequencies and dampings. In this work, rather than choosing to use some simplified mathematical equivalent of sound generation, the authors here decided to take the approach of accurate simulation of the sound generation. This task is accomplished by analysing the surface motions of objects that are animated using a deformable body simulator and isolating vibrational components that correspond to audible frequencies. The system then determines how these surface motions will generate acoustic pressure waves in the surrounding medium and models the propagation

of those waves to the listener. For the modelling of the objects, the authors chose to use the same non-linear finite element method used in [REF]. The reasoning for this choice is again similar to O'Brien's other paper in that the physical representation requires a complete volumetric model of each object, rather than just a surface approximation like in standard visual polygonal models. The key difference however is that rather than using the finite element representation as a means of pre-computing sounds for the object, the authors use the finite element to compute vibrational components in real-time as the simulation progresses and hence remove the need for expensive pre-computation.

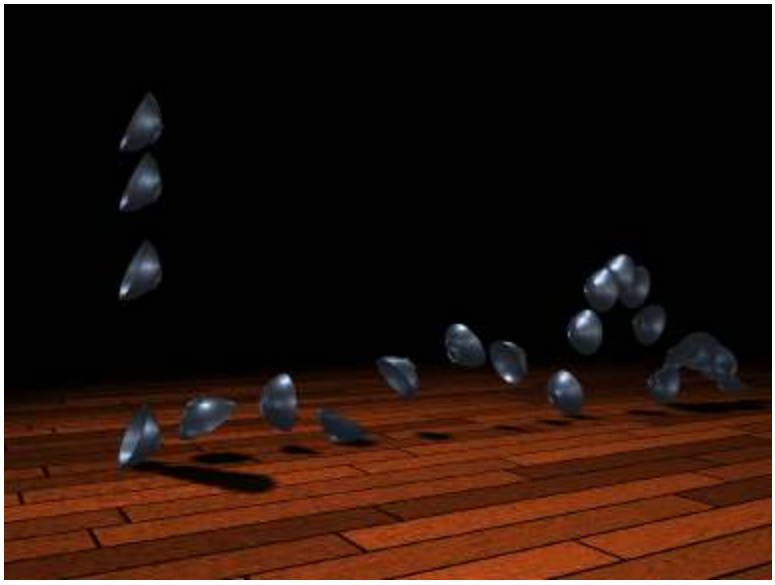

Figure 3.4 - *Demonstration of deformable models*

Having found the appropriate method for modelling the objects, the next step is to determine how their motions and collisions will affect the pressure in the surrounding environment. While it is necessary to have the complete object modelled in order to calculate the deformations in its shape it is only the surface which is of relevance when determining the subsequent pressure effects on the surrounding medium, greatly simplifying the calculations in the process. The pressure values are calculated for each triangle on the surface of the object using the following equation:

$$p = z \cdot v \cdot \hat{n} = z \cdot \frac{1}{3} \sum_{i=1}^{3} x_{[i]} \cdot \hat{n} \qquad (3.3)$$

where v and n are the velocity and unit normal of the specific triangle being evaluated, z the air's specific acoustic impedance and x the velocity of each node on the triangle in question. The resulting variable p then tells how the pressure over a given triangle fluctuates. This however will give a broader range of frequencies than is required to produce audible sounds so a process is required to remove the superfluous data. Two filters are used to remove this data: low-pass and DC-block. The low-pass filter is applied to remove the high frequencies which would otherwise cause aliasing artefacts in the final sound and the DC filter is used to remove any constant component and greatly attenuate low-frequency elements.

Once the pressure distribution over the surface has been calculated, the next step is to compute how the resulting wave propagates outward from the object towards the listener. There are a number of ways of doing this, but the authors describe the more efficient solution as being an implementation of Huygens's principle for determining the shape of a wave. A number of simplifications are made as well, including ignoring most environmental effects such as reflections and diffractions.

While this does appear to be a highly successful approach to synthesizing sounds, it is also a very complex and computationally expensive one. Even allowing for that, there is still the matter of the use of deformable models. This is a good way to accurately model the physical system, but it is not suitable for use with today's physics engines. Deformable models have recently become a part of the modern commercial engines such as Havok, but the technology is still in its infancy and is neither fully developed nor its efficiency optimised. Another limitation of the commercial engines is highlighted by another of the papers criteria: temporal resolution. In order to adequately simulate the physical sounds, the simulation needs to have an integration time-step no larger than approximately 10^{-5}s, or else it will be unable to achieve high frequency sounds. Having the physics engine running at such a high rate would put a strain on the processor and

take time away from other elements of the program. As the use of one of these types of commercial engines is a requirement of this project, it makes the above methods unsuitable.

3.4 Mathematical Derivation

The final approach to be examined will be the mathematical derivations of the modal data as detailed in (van den Doel & Pai 1996). In this paper the authors have chosen a purely mathematical approach to obtaining the modal data. They observe that sounds that result from objects being struck are a combination of a brief transient or click, with a complex frequency spectrum, followed by a relatively sustained sound, which decays over time. Also noted is how this initial click plays a vital role in the identification of the sound. There are a number of lengthy steps to the implementation of the procedure outlined in the paper, but since they are quite mathematically intensive only the theory will be covered here. A similar, yet simpler, derivation approach is also taken by Cook in (Cook 2002), which was the method adopted in one of our test applications described in section 5.3.

To begin with, a framework is introduced for the modelling of the vibrating objects. This allows the calculation of the mode frequencies of the object. The paper itself only deals with a general structure and shows the derivation for a rectangular membrane, but the authors explain how it can be adapted to incorporate any arbitrary shape. It should be noted at this point that, unlike the finite element methods described previously, this implementation only deals with the surfaces of objects. Equations are derived to describe these surfaces and their deviations from equilibrium, resulting in an equation for the time-averaged energy of the vibration. This equation takes into account the shape of the object and also it's mass density and can make certain simplifications to it based on the assumptions that the materials are homogenous and the mass is uniformly distributed across the object.

Following the discovery of the mode frequencies come the calculations of the amplitudes of each of those frequencies. Where the previous step was quite general and only determined what the possible frequencies would be in all collisions, this step takes into account the point of impact on the body and also the force of that impact. Once again the equations are generalised for all objects, but assuming the first step generated an equation for the particular object in question, then a subsequent equation can be derived here to evaluate the amplitudes for a specific subset of the frequencies obtained in the first part. This combination of equations is linear and only caters for small vibrations in the objects as larger vibrations cause non-linear results which will cause unpredictable results.

The final steps of the procedure involve how the sound will actually present itself to the listener. A number of environmental factors are taken into consideration, such as the position of the observer (which has a bearing on which modes are in phase with each other) and any other effects like echoes which would add to the original sound. For our purposes though, we are only interested in how the original sounds are formed, not how they are altered by their environment. Lastly there are the factors governing the duration of the sounds, otherwise known as damping. The key component in this is the internal friction of the object, which is a property of the material the object is composed of.

As with the many before it, this paper uses pre-computation to obtain the frequencies for the modes. Also, from the short list of objects that the authors have said they derived equations for, it does seem that a great deal of work would be involved in calculating the necessary data for any object more complex than a sphere or a cube.

3.5 Other work

The four categories described above are not the only possible approaches to modal synthesis. In this section some of the other papers that have been written in the area will be described briefly.

Moving away from methods that rely solely on the geometry of a body to determine its modes, an interesting alternative is proposed in (Djohorian 1999). Here the author represents the material of a body as a constitutive equation, i.e. an equation that is independent of the geometry of the body and depends only on its material nature. This equation is expressed as a function of stress (force per unit area) and strain (fractional change of size) and the author explains how to utiliise this equation for the purposes of mode generation.

Some other papers take a broad look at the whole sound production process and only describe synthesis techniques in passing. One such paper is (Takala & Hahn 1992), where the authors chose a form of additive synthesis, somewhat akin to modal synthesis. The natural vibration modes of a body are calculated (there is no reference to this being real-time or pre-processed) and from that, a complex vibration can be calculated using a weighted sum of the modes.

3.6 Summary

Throughout this chapter a number of procedures for implementing modal synthesis were examined in detail and although none of them matched the criteria of this project fully, they still give a good insight into the methods available to us. The primary concern with the above papers was the existence of pre-processing in nearly all of them. As already described, the work done with the finite element methods in (O'Brien et al. 2001) is most likely the most suitable method to the needs of this project. However, its substantial pre-processing time is a major hindrance in this situation. And while the mathematical derivations described in [REF] appear to be theoretically possible for any object, the sheer complexity of solving the equations makes them unfeasible for general use. The next chapter will describe in detail the approach chosen for the implementation of this project, including any difficulties and limitations the selected approach highlighted.

Chapter 4

API Design and Implementation

In this chapter we will be incorporating the research conducted in the previous chapters and demonstrating how a software API was constructed to implement the chosen synthesis algorithm, modal synthesis. We begin with a broad overview of the constructed system and follow that with a more detailed look into the particulars of how the classes and functions work. After that we describe the steps necessary to integrate the API into an application and finish with a look at how the API could be improved in the future.

4.1 Design Goals

Here we outline the design of the API which we constructed. The primary design goals of this are twofold:

1. To provide a means of testing the effectiveness of modal synthesis within the context of physically based simulation.
2. To provide a tool that developers can use to add modal synthesis functionality to their applications.

The precise functionality that we have provided is as follows:

1. The developer can designate objects within the simulation environment to be synthesis objects.
2. The developer can then associate modal information with these objects. This information takes the form of an array of triples {*m, a, d*}, where
   - *m* - frequency of mode
   - *a* - amplitude of mode
   - *d* - damping co-efficient of mode
3. The API then automatically uses the information provided by the collision detection system to drive the modal synthesis system in real time based on this.

The following is a UML class diagram of the system:

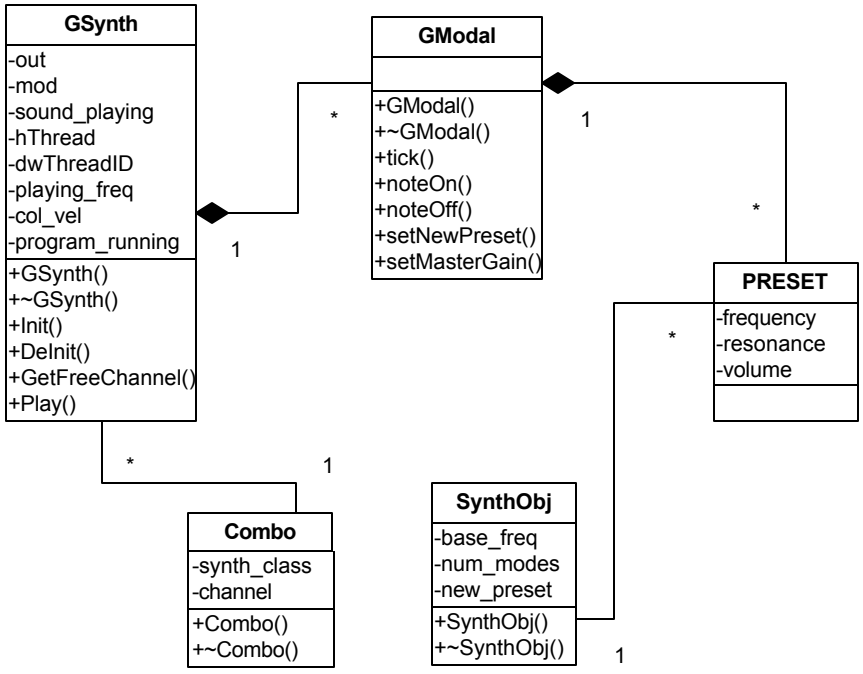

Figure 4.1 – *Class diagram for the API*

The remainder of this chapter describes in detail the issues involved in implementing the classes of the API. In the following chapter we will concentrate on describing its use in two test applications and also evaluate its effectiveness.

4.2 Overview of Implementation

The implementation aspect of the project described herein is a physically based sound synthesis API for use in any computer game or virtual environment. By using the API the developers of physical simulations are given the ability to add realistic sound to such a simulation. The system uses the standard data outputs of the physics engine as the control parameters for the synthesis algorithms.

In order to apply the theory developed up to now to the creation of a sound synthesis engine we need to implement software components which can be integrated into any desired application. The engine is implemented as an API which runs in tandem with a physics engine capable of rigid body simulation and provides an accompanying soundtrack to the physically modelled environment. The presence of a physics engine is one of the few prerequisites for our API to function, but is vitally important as the calculations and resulting data already present in the physics engine are used as the inputs to the sound engine. The way in which the API interacts with the physics engine is essentially the same way that the physics engine interacts with the graphics and control components of the application. Although most modern physics engines can perform a wide variety of simulation techniques, such as rag-doll and deformable bodies, the only component that is necessary to the operation of our system is rigid body simulation.

Due to the way in which the API was designed and created it is not restricted to any particular physics engine or any other proprietary component other than the DirectX system. Along with OpenGL, this is an accepted standard in modern games and virtual environments on Windows platforms, and exists on virtually every PC that would be used to operate such a system.

Before the API can be used it must first be initialised, achieved by calling the relevant function in one of the API classes. This initialisation function performs a number of tasks, including generating all the necessary threads and creating the links to the

DirectSound system for sound output. A second stage of initialisation must also be undertaken, but this is not automated and relies on the developer to implement it. The developer must somehow create links between the physical simulation and the synthesis system. For each object in the physical simulation which will be using the synthesis engine to produce sounds a *synthesis object* is instantiated for it. This object contains information about the material properties used on the object which is the primary influence in determining the sound that is produced. This object does not contain any functionality other than that used to access the data, so it basically serves as a store for information.

While the original aim of the project was to produce a system whereby the developers would not need to undertake any sort of pre-processing in order to use the synthesis engine, it was decided during the implementation to provide a little more flexibility. Instead of providing just the one method of obtaining the modal data for synthesis the API was designed so that a number of methods could be implemented for the gathering of modal data. The creation of these data gathering functions is left up to the user, with a data structure provided for passing the data into the API.

Once all the initialisation has taken place, the simulation is able to begin its main loop. The synthesis engine sits idle in memory until called by the main program. The only time it is actively running is when it is called by the physics simulation and while it is playing a sound. All the threads are paused until required to synthesise a sound and once a thread has finished iterating through the play loop it reverts back to paused mode once again. The next section will examine the constructed API in more detail, including the inner workings of each of the classes and functions.

4.3 Software design

The API was developed in C++ in order for it to be compatible with the existing technologies used in these areas. C/C++ is the language used in almost all computer game development, due to its speed and versatility in high-performance situations.

Applications developed on one platform using C/C++ are not generally portable to other platforms without extensive restructuring and rewriting of sections of the program. A number of components of our API, such as the multithreading aspects, contain code that is dependent on the Win32 platform and as such will not be portable to other platforms. However, as the majority of games development and physical simulation is done on the Windows platform this is not a major concern.

The API is developed to use the DirectSound API for its sound output, part of the Microsoft DirectX system.

There are three main classes in the API; the GSynth, SynthObj and GModal classes. Almost all of the components of the API are contained within these three classes, the exception being the `PlaySynth` thread function, which cannot be a method of a class and instead must remain as a standalone function. A fourth class does exist in the API called Combo, but it is only used for internal storage purposes and is not intended for general use, its specific tasks will be explained in a later section. The primary class is GSynth and it is from here that the synthesis API is initialised and operated. In the subsequent sections each class will be described in detail, outlining each class method and variable and the role each plays in the functioning of the API.

4.3.1 GSynth Class

The GSynth class is the central class in the API and the point from which the synthesis engine is controlled by the controlling application. Of the classes that the developer can use directly this is the only class in the API with any functionality built into it, the remainder of the classes being used for data storage.

GSynth Class

```
GSynth()
```

Purpose: Default class constructor

`~GSynth()`

Purpose: Default class de-constructor

`void Init()`

Purpose: Initialises the synthesis API, creating all the necessary objects and threads for later use.

`void DeInit()`

Purpose: Shuts down the API, closing any remaining threads and deleting previously created objects.

`int GetFreeChannel()`

Purpose: Finds the first free sound channel if one is available.

Returns: Positive integer value indicating available channel otherwise returns -1 indicating no free channels.

`void Play(int channel, double col_vel, SynthObj *so)`

Purpose: Plays a synthesised sound through the specified channel using the supplied data.

Parameters: `channel` – index of channel to play sound through

`col_vel` – velocity of body at time of collision

`so` – synthesis data of object involved in the collision

`RtWvOut *out[MAX_CHANNELS]`

Purpose: Realtime audio output variables, used to play back the sounds.

`GModal *mod[MAX_CHANNELS]`

Purpose: Modal synthesis objects, for generating the sounds as needed.

`bool sound_playing[MAX_CHANNELS]`

Purpose: Records which channels are currently free or playing sounds.

`HANDLE hThread[MAX_CHANNELS]`

`DWORD dwThreadID[MAX_CHANNELS]`

Purpose: Both arrays needed by Windows to track the status of threads.

`double playing_freq[MAX_CHANNELS]`

Purpose: Records what fundamental frequency each channel is to play at.

`double coll_velocity[MAX_CHANNELS]`

Purpose: Records the collision velocity of the collision using each channel.

`bool program_running`

Purpose: Indicates whether the main application is still executing.

In this class the constructor and destructor are only implemented as a formality and to adhere to standards, the real initialisation and deinitialisation being left to the `Init()` and `DeInit()` functions. This is done to give the developer control over whether the synthesis system is active in memory or not, without having to destroy the objects. `GetFreeChannel()` is used to establish if there are any channels, i.e. threads, that are not playing sounds at any moment in time. The function achieves this by checking each of the elements of the `sound_playing` array and returning the index of the first available one it finds, or returns -1 if all are in use. The last function, `Play()`, is arguably the most important one in the class. It is from here that the modal data and other physical body information is passed to the thread. When called from a program, one of the preceding statements will already have acquired a free channel number from the `GetFreeChannel()` function and will send that back as a parameter along with the details of the collision and synthesis data of one of the objects involved in the collision. The function can then resume the relevant thread and have it play the sound resulting from that single collision.

The class variables are all arrays with the same number of elements as there are threads in the system. The first two variables, `out` and `mod`, are the two synthesis components for playing the sounds and synthesising the modal synthesis respectively. They are only pointers here as the objects are not created until the class is initialised and the threads started. The `sound_playing` variable is a Boolean array which tracks which of the threads are playing sounds at any moment in time and which are free for use. `hThread` and `dwThreadID` are two variables required by Windows to control the multithreading. `hThread` is used in the `DeInit()` function of this class to close down each of the threads once they are finished running. `dwThreadID` is not used in our implementation, but is still required by Windows. The variables `playing_freq` and `coll_velocity` are used for the storage of the frequency of the sound to be played and the instantaneous velocity of one of the bodies at collision. The last variable, `program_running`, serves as a flag to specify whether the program is still active or has indicated that it is shutting down. This is checked by the threads each time they are called.

4.3.2 SynthObj Class

The SynthObj class is the implementation of the previously mentioned synthesis object, containing storage space for all the relevant modal data. Since this class is little more than a data store there are no member functions for data processing or similar functions. As will be explained in more detail later on, all elements relating to the generation of modal data are left to the user so as to provide the flexibility to obtain the data from any source.

SynthObj Class

`SynthObj()`

Purpose: Default class constructor.

`SynthObj(double bf, int nm, PRESET np)`

Purpose: Overloaded class constructor for initialisation with data.

Parameters: *bf* – base frequency of object

nm – number of modes to be calculated

np – actual modal data used for synthesis

`~SynthObj()`

Purpose: Default class de-constructor.

`double base_freq`

Purpose: Fundamental frequency of the object.

`int num_modes`

Purpose: Number of modes that the modal data contains on this object.

`PRESET new_preset`

Purpose: Modal data for the object.

The only three functions in the class are constructors and destructors. As with all the classes, the default constructor and destructor are present, as is a second constructor for instantiation with arguments for member variable initialisation. The three arguments taken by the constructor are the frequency the sound is to be played at (`bf`), the number of modes the sound will contain (`nm`) and the actual modal data to synthesise (`np`). The class variables are the same as these arguments. The variable type PRESET listed here is simply a data structure which contains all the modal data for a particular object (frequencies, amplitudes and gains). The actual variable sub-types and names can be seen in the source code appendix.

4.3.3 Combo Class

This class is merely a container for passing two variables into a function which only accepts one argument. In order to create a thread in Windows it is necessary to supply the thread creation call with a function that will serve as the body for the thread. This function must take the form of:

```
DWORD WINAPI ThreadProc( LPVOID lpParameter );
```

Any variable or class of any type may be passed as a parameter to the thread function, but no more than one is allowed. In order to successfully implement the multithreaded system, the thread functions need access to all the member variables of the GSynth class that controls the threads. Each of the threads also needs to know which channel it is responsible for playing, so this data must also be sent. Hence the Combo class was created to contain a pointer to a GSynth class and an integer variable. When creating the thread a Combo class is temporarily created into which a pointer to the main GSynth class (obtained using the `this` keyword) and the relevant channel number are stored. The new Combo class is then passed as the argument to the new thread function.

Combo Class

`Combo()`

Purpose: Default class constructor.

`Combo(GSynth *gs, int ch)`

Purpose: Overloaded class constructor for initialisation with data.

Parameters: *gs* – main GSynth class

ch – current channel to be used

`~Combo()`

Purpose: Default class de-constructor.

`GSynth *synth_class`

Purpose: Pointer back to the primary GSynth class (API entry-point).

`int channel`

Purpose: Channel index through which the sound will be played.

These three functions are standard constructors and destructors. The default constructor and destructor are present, as is a second constructor for instantiation with arguments for member variable initialisation. The only two member variables are the aforementioned GSynth class and integer value for channel. As with the previous classes these variables have been made publicly accessible to simplify the code rather than using access functions.

4.3.4 GModal Class

This class contains a set of low-level methods for supporting the actual sound generation aspects of the modal synthesis engine. Perry Cook (Cook 2002) provides a set of methods suitable for this. These provide the basic low-level functionality needed to synthesis a sound based on a given frequency, amplitude and gain, and we have incorporated these methods into our GModal class. These methods are:

- tick
- noteOn
- noteOff
- setMasterGain

To this we also have included a method, `setNewPreset()`, to register the modal data with the class. The function takes a single argument in the form of a PRESET object. This object is a simple structure which contains the frequencies, resonances and gain

values for the sound to be synthesised from. The data-type MY_FLOAT used below in the listed function definitions is actually the same as a normal `float` variable type, but as the Cook's original API was written for both Linux and Windows there are compiler macros to establish different data types for each operating system. As this class extends from another, only the new functions and those directly relevant to our own API will be described here.

GModal Class

```
void noteOn(MY_FLOAT frequency, MY_FLOAT amplitude)
```

Purpose: Informs the class that a sound is to be generated at the specified frequency and amplitude

Parameters: *frequency* – fundamental frequency for sound.
amplitude – overall amplitude for sound.

```
void noteOff(MY_FLOAT amplitude)
```

Purpose: Informs the class that the sound is to begin decaying to silence at the rate specified.

Parameters: *amplitude* – rate of decay in amplitude for sound.

```
MY_FLOAT tick()
```

Purpose: Synthesises the next audio sample for playback.

Returns: Next audio sample for playback.

```
void setMasterGain(MY_FLOAT aGain)
```

Purpose: Sets the overall gain value for the synthesised sound.

Parameters: *aGain* – overall gain value for sound.

```
void setNewPreset(PRESET data)
```

Purpose: Registers the new modal data with the class for later synthesis.

Parameters: *data* – Data structure containing the modal data for synthesis.

When a sound is required to be synthesised, the `setNewPreset()` function is called first to register the relevant modal data with the class. To first create the sound, the `noteOn()` function is called to inform the class that a sound will be generated next. This function merely registers intent to generate a sound, to actually synthesise the sound the `tick()` function is required. A call must be placed to this function for each successive sample to be generated over the course of the entire sound, so incorporating this into an iterative loop is the most effective method of execution. After a sufficient period of time has passed the `noteOff()` function can be called to being the decay of the sound. As with the `noteOn()` function before, this only informs the class of the intent and the `tick()` function is still required to generate the remainder of the samples to bring the sound to silence.

4.3.5 PlaySynth Function

One of the most important components of the API is actually the one piece that is not contained as a method of a class. The PlaySynth function is the previously mentioned function that is used as the main body to the threads. Due to architectural limitations of the Win32 multi-threading system this thread function cannot be a member function of a class. Hence the function in our API was forced to remain an external one. However, the desire to have the function as a member of a class is purely to maintain object oriented standards throughout the API and since the functionality remains the same regardless of the location the system encounters no performance decrease as a result. As this is the function where the synthesis and playing of the sound takes place, it is important to go through its inner workings in detail. To begin with, the function takes in the void data argument and recasts it back to a pointer to a Combo class, as this would be the data type originally sent to the function. This pointer is then be used to extract the GSynth class and the channel number from the Combo class. With this data now in more manageable variables there is no longer a need for the Combo object, hence it is deleted.

```
DWORD WINAPI PlaySynth(LPVOID data) {
  Combo *cb = (Combo *)data;
  GSynth *gs = cb->synth_class;
  int chan = (int)cb->channel;
  delete cb;
```

The remainder of the code in this function is the infinite loop that the thread runs in until it is terminated by the main program. The first action the thread takes when it is created is to pause itself. There are only two situations when any of the threads are needed to be active and one of them is when they are processing / playing a sound, so upon creation they need to be paused until required for audio output.

```
  while(true) {
    SuspendThread(GetCurrentThread());
```

The second case for a thread being active is when it is required to end its lifecycle. When the thread is resumed from sleep it checks the class variable `program_running` to see if the main program is still active. If it is then the thread was resumed in order to synthesise a sound so it will continue to that section of the code, but if not then it is required to shut itself down. Before doing so it must free up any memory that may have been used by the thread. Although the objects it is freeing up were not actually created within the thread they were created exclusively for use in each particular thread so it is more straightforward to delete them here rather than waiting until all the threads have ended and then free up the objects from the main GSynth class.

```
    if(!(gs->program_running)) {
      delete gs->out[chan];
      delete gs->mod[chan];
      ExitThread(0);
    }
```

If the thread function did not end, then a sound is required to be synthesised and that is what the following code achieves. The first step is to set the master gain, or volume

level, for the sound. Van den Doel and Pai (van den Doel & Pai 1996) suggest that the correct way to set the master gain for a sound in this context is to make it proportional to the normal relative velocity of the bodies involved in the collision. We follow this practice here.

```
gs->mod[chan]->setMasterGain(sqrt(gs->col_vel[chan])/10+0.25);
```

At this stage the new modal data for this particular sound will already have been entered into the GModal object that corresponds to the channel being used, so the next step is to actually go ahead and produce one of those sounds. Being based on the STK by Perry Cook our API uses the same format for creating modal synthesised sounds. The sounds are created from one of the GModal objects stored in the `mod[]` array by accessing the `noteOn()` function of one of those objects. This function takes two parameters, the base frequency of the note to be played and the amplitude.

```
gs->mod[chan]->noteOn(gs->playing_freq[chan],0.2);
```

The `noteOn()` function only informs the GModal object that a sound is about to be created, it is in the loop that follows where the process of generating the sound is undertaken. During each iteration of the loop a new sample is synthesised and played. To create each of these samples the `tick()` function of each of the GModal and RtWvOut objects are used. This function informs each of the objects that the next sample is to be created or played (depending on whether the object is an instance of a synthesis or playback class). In the case of GModal object the next sample will be synthesised according to the parameters set by the preceding function calls. This sample data is then passed as an argument to the `tick()` function of the RtWvOut object which will play the sample on the default audio device. A special case iteration of the loop can be seen here where, at 20,000 samples, an additional function call is made to `noteOff()` to tell the synthesis algorithm to begin decaying the sound to silence. The single floating point argument is the amplitude of the decay rate. The loop then continues as normal with the sound decaying to silence. It was decided that 50,000 samples were enough for any

sound to be played, as this was the duration of a synthesised vibraphone sound which had a high resonance value.

```
    for(int i=0; i<50000; i++) {
      if(i==20000)
        gs->mod[chan]->noteOff(0.5);
      gs->out[chan]->tick(gs->mod[chan]->tick());
    }
```

Once the sound has finished playing the channel can be marked as free again.

```
    gs->sound_playing[chan]=false;
  }
```

The return statement is never actually reached at any time during the running of the program as it was decided to use the ExitThread function to close the threads. However it is still required for compatibility with the multithreading format, so it is left in.

```
  return((DWORD)data);
}
```

Now that all of the classes and functions have been examined, the following section will be used to illustrate how these classes can be integrated into an application.

4.4 Using the API

In order to use the API as a component of a physical simulation a number of steps must be taken to set up the system for use. The entire API is centralised in one area and only a single C++ include file (GSynth.h) is required to integrate the API classes and functions into the program.

```
#include "GSynth.h"
```

Once the include file has been specified, the first step to setting up the API is to create an instance of the GSynth class. As this is the crux of the synthesis system and will be used in a number of locations throughout the program it is advantageous to create this as a global variable.

```
GSynth sounds;
```

The above class doesn't contain a functional constructor or destructor and has to be initialised and de-initialised manually by the developer. This was decided so that it would be possible for the developer to enable and disable the synthesis engine at will, without having to actually create and destroy the class object to do so. Hence, the initialisation routine does not have to be called in any particular place as long as it takes place before the engine is used to synthesise and play sounds. The necessary function is called as follows:

```
sounds.Init();
```

There is another stage to the initialisation, although this segment can actually take place regardless of the activity status of the engine. As previously detailed, a synthesis object must be created for each of the objects in the physical simulation that will be used to create sounds. The SynthObj takes three parameters in its constructor, the fundamental frequency of the object, the number of modes and a PRESET structure containing all the modal data for the object.

```
SynthObj *so = new SynthObj(200.0,4,new_preset);
```

In this particular use of the SynthObj class the objects were allocated dynamically. This however is not a requirement for their use, but merely a consequence of the way in which they would be used later in the framework we created for testing. It is also not necessary to include all the object data as arguments during creation, as all the variables are

publicly viewable and can be edited in subsequent statements. In order for the API to be flexible enough to work with any generic physics engine and not be tied down to the specific functionality of one commercial product there is no predetermined link between the created SynthObj objects and the instance of the GSynth class. We shall see some examples in chapter 5 of how this can be accomplished in our test applications. In our framework implementation, the rigid bodies in the Havok libraries contained a void pointer storage area which allowed for any single item of data to be assigned to the object using code similar to the following:

```
rb->getPrimitive(0)->setUserInfo((void *)(so));
```

Once all of those elements have been set up, the system is ready for use. As with the creation of the SynthObj components, this next part is very much implementation specific, but we will merely highlight the elements that will be common to all implementations. Once the physical simulation has reported that a collision has taken place the first thing that must be done is to determine if a free channel is available. This is achieved as follows:

```
int chan = sounds.GetFreeChannel();
```

If the above function returns a positive number to indicate the ID of the first free channel it found, then the program can proceed to create a synthesised sound. In order to be able to tell the synthesis engine what sort of sound to create and play, we must first establish what type of objects were involved in the collision. Knowing that, we can then find the SynthObj object that corresponds to the rigid body which will be making the sound and send it along with other relevant information to the API to create the correct sounds.

```
sounds.Play(chan,ce.m_nrv,(SynthObj*)(rb->getUserInfo()));
```

Some of the elements of this illustration will vary slightly depending on the particular software libraries that are being used, but the basic sequence of events and functionality will remain the same. Having finished looking at how the system was finally constructed

and operates, the next section will deal with some of the obstacles encountered during development.

4.5 Difficulties Encountered

Throughout the development of the API, a number of issues arose that impeded the progress of the application's design. These ranged from common problems, such as apparent incompatibilities between the functionality of different components being used, to problems which caused us to re-examine the approach being taken towards implementing the program. The following sections will describe some of the larger problems in detail.

4.5.1 Control Parameters

One of the problems first encountered in developing the synthesis procedures was how to translate the data obtained from the physics engine into something that could be passed as parameters to our API. The data which is readily available about each object in the physics engine includes mass, volume, instantaneous velocity and other information directly relevant to the simulation. When a collision occurs, some additional information becomes available, consisting of the IDs of the two objects in the collision, the magnitude of the force of the impact, and the point in the simulated world where the collision occurred. It should be noted that while it is entirely possible that more than two objects could collide together in a single moment, each contact between two of the bodies is treated as a separate collision for simplicity. One problem in the same area was the fact that the lower-level internal workings of the API were simplified to only deal with simple objects, specifically solid, cuboid "bars" of finite length. This is not to say that only bar-shaped objects can be modelled, as any object can have synthesised sounds as long as the necessary data has been obtained. Instead it means that the algorithm uses a bar-shaped object to represent the actual object during some calculations, but still uses the correct modal data that was provided. One of the parameters which utilised this simplified

format was the strike position. This operand is a floating point number between zero and one, illustrating where along the length of the bar the strike took place. No mention is made to any other dimension of the object apart from the length, which leads to a number of assumptions being made about the object. Firstly it assumes that the object is perfectly uniform along the described axis, both in weight and shape. Any deviation in either of these values at some point along the object's length would result in a different sound being created rather than simply the small differences the sounds make the closer they are to either end. At its current stage, the only control parameter used in the synthesis, other than the modal data naturally, is the force of the impact, or in the specific case of using the Havok engine, the normal relative velocity.

4.5.2 Synchronisation and Multithreading

Having developed the algorithms for the synthesis techniques was a good start, but at that stage it was only possible to play a single sound at any one time. It was determined that multiple simultaneous sounds would not be possible without some form of multithreading. In our specific implementation of the API the Havok physics engine already had multithreading built into it and was using it for event reporting, so adding our code in as an extension to theirs was considered to keep the any additional overheads to a minumum. This proved to be an unfeasible solution due to the massive relative difference in the update frequencies used by the physics simulation, compared to those required by the sound synthesis algorithms. Computer graphics are usually shown at frame-rates between 30 and 120 frames per second. The physical simulation will be executed at a moderately higher rate to maintain a smooth and accurate flow for the animation. High quality sounds on the other hand, can be played at frequencies of anything up to 44,100 Hz. When an attempt was made to run the synthesis components in the same area as the existing components it was found that the sounds were being played back with a sort of stuttered effect on them. The synthesis techniques were being executed correctly, but due to the low rate at which each sample was being played, there

were noticeable gaps between each one giving a type of stuttered effect. An alternative solution was determined whereby the synthesis engine would control its own threads.

Multithreading in C++ on the Win32 platform is implemented using standard function calls to the Windows API. In order to create a thread, a function must first exist which will become the equivalent of the *main* function to an application. This function does not differ in structure from any other regular function, the only requirement being that it takes a single parameter of type *LPVOID* and return a value of type *DWORD*. When a thread is created it returns a "handle" which can be stored as a variable for later access to the thread (e.g. for changing its priority, pausing it, or shutting it down) and then begins executing the function assigned to it. Although the task priority of a thread can be specified explicitly, it was deemed unnecessary for the small amount of processing each individual thread would do and also while specifying a higher priority would probably allow a greater number of modes to be calculated per object, it would be at the expense of processor time for the other components in the simulation. Once the thread has been created, a number of control functions then exist to pause, resume and stop the threads (PauseThread, ResumeThread and StopThread respectively).

Once it was decided that the synthesis engine would require its own multithreaded system the next step was to resolve exactly how to implement this system. This first approach to dividing the synthesis processes between separate threads involved calling a new thread each time an event occurred that required a synthesised sound. Once a call was made to our API informing the engine of a sound producing collision, a thread was created and the synthesis parameters passed to it. The problem with this approach was the overhead of creating and destroying the threads. Since windows opens and closes threads behind the scenes all the time, it was not believed that there would be a problem with taking the same approach. This unfortunately was not the case and instead proved to be problematic in establishing correct synchronisation between the on-screen collisions and the corresponding sounds. A number of approaches were tested until it was found that the most effective method of using the multithreading for sound was to initialise all the threads in the beginning and simply pause them when not required. A thread would be

created for each channel in the system, up to the limit set out by the developer, and then each would be paused. A global array of Boolean variables is kept in the system to keep track of which threads are active at any particular time; each element of the array is set to false unless the corresponding thread is playing a sound, in which case it would be set to true.

4.6 Future additions

It should be possible to implement an alternative method whereby the sound which has been playing the longest would be stopped and the new one would take its place. This would ensure that every sound that occurs in the simulation would be heard, but it would also mean that in a scene where the number of sounds exceeds the number of channels by a significant number, the sounds would be cut off quite early and would cause some noticeable aural artefacts. Hence it was decided to allow each sound to finish playing and just skip the others. Another option for deciding which sounds would be played would be to determine if they would be noticeably heard by the user. It has already been mentioned about how the force of the collision has a proportional effect on the volume of the sound it produces. By using the same information it can be determined whether the sound would be heard by the user or not. If a number of relatively high volume sounds are playing then it will make little difference whether one of a much lower volume is played or not. This could end up being more complicated though as the control system would need to take into account how long each of the sounds had been playing and what the actual amplitudes were at each moment in time.

Also related to the length of the sounds is determining the number of samples that should be played for each sound. It was decided that 50,000 was a good number initially as it would allow all sounds to play fully, regardless of their individual parameters. While this works well for sounds that resonate for a time after collision, it is a waste of resources for shorter sounds. It was discovered experimentally during the implementation that some of the sounds that could be produced would require as little as 5,000 samples to play in their entirety, which then leaves 45,000 being used with no justification. Considering the

length of the sound is directly related to the resonance factor it may be possible to construct a formula which would determine the optimum number of samples for each sound, thus freeing up each channel quicker and allowing other sounds to be played.

One of the characteristics of the modal synthesis algorithm is that only one of the bodies involved in the collision is taken into account when actually synthesising the resulting sound. Naturally, in the real world when two objects collide, the properties of both bodies are relevant to the sound produced. We experimented with the idea of generating two separate sounds (one from each of the objects involved) and playing them simultaneously, but they merely sounded like two sounds being played together, rather than merging together to produce a single sound. A new algorithm would be required in order to implement this feature correctly.

Chapter 5

Testing and Evaluation

Having examined the workings of the API in detail in the previous chapter, we now describe the application framework used to assist the implementation and to later test the finished system. Following that, the specifics of both the test applications built from the framework will be described, including information about the sources of the modal data used in each program. The first application models a musical instrument in an enclosed room with all data being pre-determined; the second modelling a simpler membrane structure with much of the data being calculated during the execution. Lastly the fully implemented system will be evaluated from the perspective of both the user and developer.

5.1 Implementation of Testing Framework

In order to assist in the development of the API some form of framework application was required which could use the functions of the API. This program also doubled as a good basis for building the test applications to evaluate the success of the implementation later on. The program developed for these purposes takes the form of a visual simulation written in OpenGL and containing the Havok physics engine. Both the framework and the Havok library are written in C++ and utilise the object orientated methodology, thus assisting in creating the API to follow the same methodology. The Havok libraries contain tools to allow for the creation of simulated worlds using a 3D modelling package, such as 3DS Max or Maya, and then importing them into any program to be simulated.

Using the provided tools and the documentation it became possible for us to create a program which would allow the modelling of any arbitrary environment and test the sounds in it. Naturally, since the Havok system provides a method for exporting physical simulation data from 3DS Max it also provides a set of functions for importing this data into the developer's program for easy creation of a simulated world. This import feature is for reading in the geometry of rigid bodies and any other physical constraints that may exist in the system, but at the time of development at least, there was no straightforward way of obtaining the graphical counterparts to these objects. As such, a second importer had to be written which would parse the Havok data file after the main import had finished and extract all the graphical geometry from the file. Each of the objects in the Havok world were then paired up with their counterparts in the graphics and stored in a linked list.

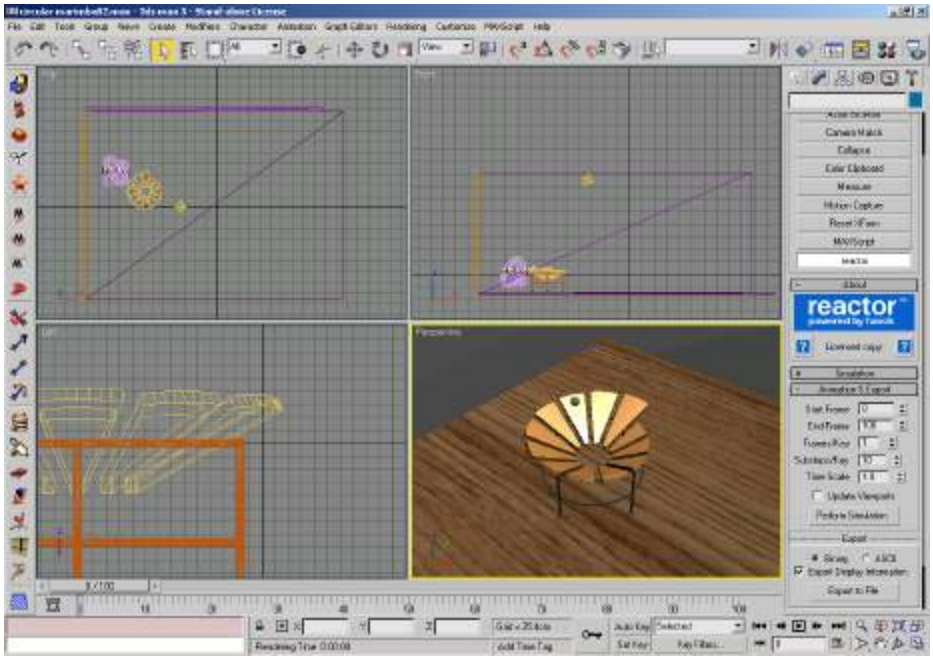

Figure 5.1 - *Using the Havok Reactor system in 3DS Max*

5.1.1 User Interaction

Each time the program goes to update the scene visually it first processes the user input, both for the viewer controls and any interaction with the simulation. The user has complete control over the camera in the simulated world, and can move around the virtual room to see the instrument from all angles and positions. The controls chosen for this were the same as those for the default settings on most first-person shooter (FPS) games out today for the PC. The keys W, S, A, and D are used for movement in the four compass directions and the mouse is used for rotating the viewpoint. All the directional keys are used relative to the viewpoint, so pressing forward or backward will move along the viewing axis. Two additional user controls exist which allow some limited interaction with the simulated world. Pressing F1 adds a ball to the scene at a random location above the scene. There is no way to directly control any of the objects once they have been added to the scene. The balls are added at locations above the environment so that they will fall under gravity, and the random positioning also ensures that a set of balls will have different velocities when they first strike an object and that each one will strike a different sequence of objects. In addition to the ability to add objects to the scene, it is also possible for the user to remove all the previously added balls using the F2 key. The more objects there are in the scene the greater the computational load on the CPU to maintain all of them and also maintain a stable visual frame rate, so it is useful to have a way of resetting the simulation. The final control available to the user is the ability to pause the simulation, simply implemented by pressing the PAUSE key. It is worth noting that this only pauses the physical simulation, not the whole program, so the user is still able to move the camera around the temporarily static environment.

5.1.2 Operational Details

Following the processing of the user input the program makes a call to the Havok system informing it how much time has passed since the last time the scene was updated and to forward the simulated world by that amount. During this world update, the simulation will also determine if any collisions have taken place between any of the rigid bodies. If any such collisions do occur then the pre-determined collision event call-back function

will be called once for each collision, in order to handle the outcome. Parallel to the collision detection occurring, and the subsequent sound synthesis and playback, the program then goes through every rigid body object in the scene (as stored in the linked list) and acquires the position and orientation for them so that the visual aspect of the simulation will match the physical part. Each of the rigid bodies in the environment possesses the position and orientation parameters from which the graphics component can determine how and where to draw each object on the users screen. This update loop is repeated endlessly until the user quits the program.

Using this testing framework, two specific applications were developed in order to demonstrate the versatility and effectiveness of our created synthesis API. The first application shows how the API could be used to provide a sound environment for a complex scene involving various objects of differing shapes and that exhibit different sounds. The second application was visually simpler, but the purpose there was to illustrate how all the modal data could be computed in real-time for an object and how the point of impact on a body affects the resulting sound. Both of these applications will now be looked at in more detail.

5.2 Specifics of pre-calculated data demo

The scene we used as our primary simulated world was a musical instrument situated inside an otherwise barren room. The instrument itself is not a real-world model, but it does bear musical resemblance to many percussion instruments in the real world, specifically the marimba and vibraphone. The rather unusual design was chosen so that it would provide an interesting physical environment. Each of the pie-slice shaped keys are modelled as objects of the same metal material, with each key being at a unique pitch, that varies across the twelve notes in an octave. Although this demo does vary the sounds produced by the collisions, it uses pre-calculated modal data as its inputs. As a result, it does not have the same wide range of sounds that would exist if all the data was generated on the fly. The most significant piece of data to be used from the simulation here is the force of impact. Due to the base modal data being set values at the beginning

of the program it is not possible to take into account factors such as point of impact, but while this is a disadvantage, the demo shows that it does not take much away from the general experience. The value obtained from the force of impact is used to determine the overall master gain value that the modal calculations will use. While this does appear to be little more than an adjustment of the total volume level of the sound, it does in fact have a bearing on the relative volumes of the individual modes, in particular the fundamental mode.

Figure 5.2 - *First test application, showing virtual instrument*

Figure 5.3 below lists an example of the modal data used for the vibraphone sound in the application. This data was taken from the sample data that accompanied Perry Cook's STK. The specific data listed here is used to synthesise a resonant metallic sound. Although there is only enough data for four modes, the resulting sounds were still instantly recognisable as the materials they were intended to represent. The API can compute up to 32 modes for each object, but it was discovered experimentally that four was the minimum for the sound to be both recognisable and convincing. As will be seen

with the next test application, the use of higher than four modes adds more depth to the sound, but does not produce a different sound to one using a lesser number of modes.

Frequency	Resonance	Volume
1.0	0. 99995	0. 025
2.01	0. 99991	0. 015
3.9	0. 99992	0. 015
14.37	0. 9999	0. 015

Figure 5.3 – Table of modal data for resonant metallic sound (e.g. vibraphone)

This application provided a good example of the variety of sounds that our implemented system can produce. The program can also handle a number of objects being on the screen and colliding at the same time without a serious decrease in the overall performance. Unfortunately, with only a limited set of modal data to experiment with, the range of sounds that were produced was restricted to a mere ten unique sounds.

5.3 Specifics of derived data demo

In addition to the above demonstration application a second test system was constructed. Unlike the other demo which used predetermined modal data to generate the sounds, this program generates the data itself in real-time as the collisions occur. As described in [REF – Perry Cook] equations can be derived to describe the surface vibrations of any arbitrary object. We have already seen that values for frequency, resonance and volume are required for each mode that is to be generated; hence this section deals with the outlining of procedures for deriving these values from first principles.

The equations make the assumption that the properties of the objects such as density, tension and thickness are evenly distributed across the body and are the same value at each measured point, but this is a simplification procedure that would most likely be

undertaken for the simulation regardless, so this doesn't become a limitation. For our examples we chose to model a two-dimensional square membrane as it suitably demonstrates the relevant principles without getting overburdened by the mathematics. The square membrane acts much like a two-dimensional string, but instead of merely exhibiting vibrations which are sums of sinusoidal components in the x directions, it also vibrates in the y direction. Unlike one-dimensional objects though, the square membrane modes are not integer-related harmonic frequencies. In fact they obey the following relationship:

$$f_{mn} ? f_{11}\sqrt{((m^2 ? n^2)/2)} \qquad (5.1)$$

where m and n indicate harmonics on the two-dimensional surface and can range from 1 to infinity and f_{mn} being the frequency of that particular harmonic. f_{11} is the fundamental frequency, defined as $c/2L$ (the speed of sound on the membrane divided by twice the diagonal length). We can use this formula to simulate any level of harmonic we wish. For example, calculating for f_{11}=330Hz, m=2 and n=3 will yield the following:

$$f_{23} ? 330\sqrt{\frac{2^2 ? 3^2}{2}}$$

$$f_{23} ? 2145Hz$$

The above equation (5.1) can be extended further to describe a rectangular shape as well, with the modal shapes being spatially stretched and the frequencies changing accordingly. Having the frequencies of the modes is only the first step in the derivation. We now need to know the corresponding resonances and volumes for each of these frequencies. Striking the membrane with an ideal impulse at any point would excite each mode with a gain equal to the corresponding mode shape value at that point. By representing the membrane as a unit square, the gain on each mode as a result of striking the object at an arbitrary point (x,y) will be represented by:

$$gain_{nm} \; ? \; \sin(\, n?x)\sin(\, m?y) \qquad (5.2)$$

Using these formulae as a foundation, this application can create even more accurate sounds for the physical simulation due to the ability to take into account point of collision. As shown in the screenshot below, a simple single structure was created to test the differences that point of collision can actually make on the resulting sounds. The environment is a model of a stiff membrane held above the floor at its four corners by thin rods. Although the model of the membrane does more visually represent a thicker object, this was merely to simplify the design process as the Havok engine does not handle very small measurements accurately. In the synthesis process however, the object is still being treated as a two dimensional object.

Figure 5.4 - *Second test application, showing membrane*

Since the formula for calculating the gain values requires the co-ordinates to be positive values from zero to one, we must translate the data the physics engine returns into

something more manageable. The Havok engine returns all points of impact in world co-ordinates, as opposed to being relative to the object. Normally to calculate the point of impact relative to the object we would need to subtract the vector position of the object from the returned world co-ordinates, but since we were only providing sound to one object we simply positioned that object at the origin to make the returned co-ordinates the same as those on the object. We have already mentioned how the Havok engine does not handle very small sizes well, so modelling the membrane object as a 1 x 1 square was not possible. Instead, the object was modelled at a much larger proportion and the co-ordinates scaled back for synthesis. The tables in figure 5.5 show the mathematical differences that occur from the recorded collisions at two different places on the surface of the membrane. These values are calculated using equations 5.1 and 5.2 with random locations chosen to show the vast differences that result from different positions.

Position: (0.5,0.5)	
F11 = F11 * 1.000	Gain = 1.000000
F12 = F11 * 1.581	Gain = 0.000001
F13 = F11 * 2.236	Gain = 1.000000
F14 = F11 * 2.915	Gain = 0.000001
F21 = F11 * 1.581	Gain = 0.000001
F22 = F11 * 2.000	Gain = 0.000000
F23 = F11 * 2.550	Gain = 0.000001
F24 = F11 * 3.162	Gain = 0.000000
F31 = F11 * 2.236	Gain = 1.000000
F32 = F11 * 2.550	Gain = 0.000001
F33 = F11 * 3.000	Gain = 1.000000
F34 = F11 * 3.536	Gain = 0.000001
F41 = F11 * 2.915	Gain = 0.000001
F42 = F11 * 3.162	Gain = 0.000000
F43 = F11 * 3.536	Gain = 0.000001
F44 = F11 * 4.000	Gain = 0.000000

Position: (0.87,0.4)	
F11 = F11 * 1.000	Gain = 0.377711
F12 = F11 * 1.581	Gain = 0.693291
F13 = F11 * 2.236	Gain = 0.894831
F14 = F11 * 2.915	Gain = 0.949180
F21 = F11 * 1.581	Gain = 0.233438
F22 = F11 * 2.000	Gain = 0.428478
F23 = F11 * 2.550	Gain = 0.553037
F24 = F11 * 3.162	Gain = 0.586626
F31 = F11 * 2.236	Gain = 0.233438
F32 = F11 * 2.550	Gain = 0.428477
F33 = F11 * 3.000	Gain = 0.553036
F34 = F11 * 3.536	Gain = 0.586625
F41 = F11 * 2.915	Gain = 0.377711
F42 = F11 * 3.162	Gain = 0.693291
F43 = F11 * 3.536	Gain = 0.894832
F44 = F11 * 4.000	Gain = 0.949180

Figure 5.5 – Table of modal data calculated in real-time during execution of the test application

There are a few observations that can be made about the values obtained from the two impacts. Looking at the frequency columns in both instances we can see that the values are identical in both cases. This should be expected, as the only condition under which the frequencies would change would be if the structure of the object was altered somehow. Neither our system nor the underlying physics engine accommodates such a situation, so this is not really an issue. Consequently, these values do not need to be generated at each collision, but can instead be calculated at initialisation and stored with the object in the same way that the pre-determined values are stored in the previous test application. On the other hand the values for mode gains are those that change at each collision so they need to be calculated each time. We can see here that the changes in values from moving the point of collision outward from the centre of the object can cause quite a drastic difference in the resulting data. The ensuing sounds only appear to differ slightly to our ears, despite the vast mathematical difference, but the difference is noticeable and proves that the point of impact is important to the accurate simulation of an environment.

5.4 Evaluation

In order to evaluate the success of the implementation it was examined from two points of view: the developer and the end-user. As has just been described in the previous sections of this chapter, the API is not a standalone program in itself and instead requires an additional framework in which to run, so it is prudent to examine how much work is placed on the developer to incorporate the synthesis into their own programs. Of possibly greater importance though is how the end-user will perceive the sounds being created by the system, so this will also be examined by comparing how realistic they are to the pre-recorded sounds.

5.4.1 Developer

In order for the system to be appealing to developers it must offer an incentive to move away from the established sample-based methods, so the system is required to be straightforward for implementation. We have already outlined the steps that must be taken by a developer to integrate the synthesis API into a physical simulation. While every effort was made to ensure as much code as possible was self-contained in the classes, there is still a reasonable amount of code that the developer must write to fully incorporate the API. All the actual synthesis code is encapsulated in the API itself, so the developer does not need to be concerned with the intricacies of the algorithms, but the responsibility of providing the modal data is still placed on the developer. While a sample-based engine can be used by anyone without knowing how the recordings are being played back, this implementation does require that the developer has some basic working knowledge of modal synthesis; at the very least how the numerical data will affect the overall sound.

Performance is another major issue for the developer, which in the case of our API amounts to whether the system can successfully run in real-time and does not cause too much of a drain on the CPU during execution. Sound engines using sampled sounds have been perfected over the years and now require very little time from the CPU to play back many sounds simultaneously, so our system needs to come close to equalling that performance. This has been achieved with the API, but only to a limited extent. The system can successfully synthesise sounds for approximately 10 objects simultaneously without a noticeable decrease in performance. This is only possible when pre-determined data is used. The second test application demonstrated that the additional real-time calculation of individual mode amplitudes introduced a considerable decrease to the performance level, in some cases causing the application to produce errors in the simulation. Hence it is clear that while the system can be used to provide an alternative to sampled sounds, but only for a limited number of objects and only using pre-calculated modal data.

5.4.2 User

Regardless of the onus placed on the developer to achieve the successful integration of synthesis into a physical simulation the results should be expected to be impressive for the user. Despite early issues with correctly synchronising the audio and visual components, the final version works quite well. With the sounds being synthesised as they are being played makes it easier to have the sounds starting at the correct time, rather than synthesising each one in its entirety beforehand and then playing back once complete. It is uncertain whether the sounds produced from more intricately constructed environments would provide the same convincing effects that were produced here, but going by what we have seen here, there is no reason to make that assumption.

A reasonably powerful PC is required to comfortably support an implementation of this type. A number of computers were used during the testing phase of the project, ranging from Pentium II up to Pentium IV, with memory capacities from 128MB to 512MB. While the processor and memory did play an important part in the performance of the system it was determined that the most important component was the graphics card. Naturally the synthesis engine does not use the graphics card for any of its operations, but with all the 3D graphics tasks being offloaded onto the graphics processor it left more of the main CPU for the synthesis calculations. Any modern graphics accelerator with 16MB or greater in onboard memory will be sufficient to take the responsibility of representing the virtual world on screen.

Chapter 6

Conclusion

In this chapter we conclude the thesis by comparing the aims and objectives of the project with the achievements and also the suitability of the technology for commercial use. We will begin by summarising the main issues addressed in the thesis, followed by a detailed analysis of the success of the overall project. With the benefit of hindsight, we also identify a list of enhancements and recommendations for future work.

6.1 Summary

It was identified that there was a major limitation with the sample-based sound systems currently in use in modern computer games and virtual environments and that an alternative was required. We began by outlining the specific issues present in existing audio engines, specifically the limitations that arose when physical simulation was used. Those restrictions included the invariance of the sounds used, compared to the visual dynamics provided by a physics engine. Hence it was determined that in order to match the new visual level provided by the physical simulation, the sounds would need to utilise the same technologies. Rejection of the traditional sample-based approach led to the examination of other methods of sound generation. Thus, the ultimate goal of this project was decided as the design and development of a system with which a games developer could easily add physically-controlled synthesised sound calculation to their game.

During the research phase of the project it was discovered how the synthesis of sounds had historically been used primarily for musical applications, but that the same principles could be applied quite appropriately to our intended application. We then examined a number of synthesis techniques, establishing with each one whether its particular approach to synthesis met with the criteria set out for the project. Ultimately it was modal synthesis that was chosen as the algorithm that would be used for implementation, due to its foundation on physical representations of objects and because it does not require a vast amount of calculation to operate.

Having selected the most appropriate synthesis technique, the next stage was to examine a variety of approaches that could be taken to obtain the necessary data to serve as input to the modal synthesis algorithm. These methods were studied under four categories: finite elements, deformable models, mathematical derivations and measurement. Each of the methods in these categories could provide the required data, but they did not all meet the criteria set out for the project. As such, we were left to choose from either deriving the data from first principles or gathering it via the use of special machinery that could acquire the data from actual physical models.

With all the research completed, and the most appropriate methods chosen for synthesis and modal data, we began the implementation phase. This took the form of an API which would seamlessly integrate with a standard physics engine and provide synthesised sounds for any object in the simulation. The API was also designed in such a way that it remained independent of any proprietary technologies, such as the particular physics engine that was used during development. Given that the system requires another application and a physics engine to control it, the steps that would need to be taken to integrate the API into an application were also described in detail.

Since the API could not function as a standalone program, a framework application was required for the purposes of testing the system. This was made as a physical simulation using the Havok physics engine, wherein a modelled environment could be loaded in and the user would be able to interact with the objects contained in it. From this framework

two different test applications were constructed. The first application used pre-calculated modal data to demonstrate how the API could be used to provide a rich soundtrack for a detailed virtual environment. Instead of using pre-determined data for the second application, we decided to demonstrate how the application could be used to calculate all the necessary modal data at run-time. Both of the test programs successfully illustrated how the API could be used as an alternative to sample-based systems under the right circumstances, and also highlighted some of the limitations that existed using both sources of data.

6.2 Achievements

During the initial stages of the project, a number of criteria were drawn up that the synthesis engine would need to adhere to, in order to be appealing to a developer. In this section we will describe each of these criteria and examine to what level each was achieved in the final implementation. The original criteria were:

- The synthesis method employed will need to be suitable for use with rigid body simulation.
- This method will not require any more data inputs than that normally available from a modern physics engine.
- No pre-recorded samples should be necessary for the system to function correctly.
- The sounds produced by the system will need to sound close enough to their real-world counterparts to constitute a plausible simulation.
- Neither the sound system nor the physically modelled objects should require any pre-processing before the system will function correctly.

With the rising popularity in physical simulation in computer games today, the most common use of this type of technology is with rigid body simulation. In order not to add excessive requirements onto an already burdened system it was necessary to ensure that the synthesis methods could operate fully using the level of physical simulation being

used in current applications. Although the use of rigid body simulation imposes a few limitations on how physically accurate the synthesis system can be, its minimal use of system resources is an obvious advantage. In order to fully synthesise the sounds resulting from a collision of two bodies the synthesis algorithm requires, along with the modal data for the objects, the force of impact and the point of impact. Both of these parameters are readily provided after each collision that the engine detects. Some calculations are required to be performed on this data once it has been obtained, but no extra values are required from the physics engine. The implemented system will work well with any physics engine and uses only the most elementary functions of such an engine.

After establishing the suitability of the system to the current level of support technologies available in the commercial market today, the next stage to evaluate was the sounds themselves. Regardless of whether the system is using pre-determined modal data or those derived from first principles, there are still no sampled sounds involved in the process. During our research phase, we encountered a number of synthesis methods which still used some form of sampling as a foundation, e.g. subtractive synthesis, but with modal synthesis we were able to accomplish our intended task without using any samples whatsoever. The plausibility of the sounds was one of the more difficult elements to quantify in terms of its success. With only a limited set of modal data with which to test the system, it is difficult to firmly establish whether the majority of the sounds produced by such a system would be comparable to those reproduced from recordings. However, with the data that was available to us, along with the data derived from first principles in the second test application, the sounds created were appropriate to what was being displayed visually.

The decision not to allow any pre-processing of data to occur was the one prerequisite that caused the most problems for the development. As a result of this decision, a number of synthesis methods and sources of data had to be abandoned because of the length of time they needed to operate. However, we were able to successfully achieve the goal and implemented the API without the need for any forms of pre-processing.

With the second test application we showed successfully how the modal data could be generated on request for each collision that occurred. Unfortunately, as a result of the intensive processing required for each object, we were only able to add a few sound emitting objects to the scene before a noticeable performance decrease occurred.

One of the major limitations of the API is the lack of modal data available for it. All of the modal data used in the testing of this project was obtained from the sample applications of other systems. Very few of the research papers that did not deal with the actual calculation of modal data indicated where else this data could be obtained from. It seems that future application of these techniques will have to include methods for generating this data. We suggest a possible practical approach to this problem in section 6.4.

6.3 Project Assessment

The project successfully achieved what it set out to do, by researching and constructing a system that could synthesise sounds in real-time with data supplied by a commercial physics engine. We examined a number of different synthesis methods and based on the criteria established for the project, chose the system which most closely adhered to them. Having selected modal synthesis as the most suitable algorithm, we constructed an API which implemented it.

Given the opportunity to undertake the project again, some different approaches would probably be taken. Something which should be considered is to revise some of the criteria for the endeavour, with particular regard to pre-processing. Many of the research papers, which dealt with the use of some level of pre-processing for the synthesis, produced sounds of much higher quality and suitability to the source objects than those created using on-the-fly calculations. One of the most accomplished of those methods was the use of finite-element models. Modal synthesis would still appear to be a suitable algorithm, but perhaps not the only good one. We briefly described granular synthesis earlier in the research phase of the thesis, but passed it over in favour of modal synthesis,

mainly due to lack of direct physical representation of sounds. However without the previous constraints of not using samples and no pre-processing, granular synthesis could provide a suitable alternative.

6.4 Future work

While the API has proved to be a successful implementation, using the intended technology, the system can sometimes suffer from a high load on the processor when more than a few bodies are involved in collisions simultaneously. The API was developed more as a proof of concept, as opposed to a releasable commercial product, so the code and its algorithms are not optimised. Streamlining the execution process, for example by writing the synthesis routines in assembly language, would most likely provide a noticeable performance increase.

In addition to aforementioned improvements to the API, it should also be possible to provide extra resources for the developer, such as a library of pre-derived modal data for a collection of common objects. By implementing such a resource library, it would open the system up to the possibility of using some of the previously mentioned synthesis techniques that were rejected on the grounds of there being too great a time needed to process them, e.g. finite element.

Another option to consider would be to use some sort of low-polygon proxy object for the sound synthesis compared to what is displayed on screen. It is already common practice to use level-of-detail techniques in graphics today to reduce the load on less powerful processors and now physics engines are using similar low detail models to simplify their calculations. The simplest way of executing this is to adjust the number of modes being generated for each sound.

While the capability exists to have multiple sounds playing simultaneously using the API, there is a minor issue when more than one of these sounds originates from the same object. Research in papers such as (Smith 1996) describes how the sound made by an

object which is already producing a sound is subtly different to the sound already being produced. The vibrations already existing in the object interfere with those caused by the second vibration and in most cases add together to produce a louder sound. Addressing these issues would hugely improve the quality of the produced sounds.

One final issue is that of the sound propagation and rendering. These are not directly related to the synthesis of the sounds, but they still play a vital role in the perceived realism for the listener. Factors such as the distance of the sound source and its position relative to the user and environmental effects, such as reverberation are important to the overall effect. In order to complete the immersive experience for the listener these features would need to be incorporated later on.

6.5 Conclusions

In conclusion we shall now evaluate whether sound synthesis can be considered a technology worth researching further towards its use in commercial applications in the future. When implemented correctly, it would provide a much richer soundtrack for an environment than provided by simple recordings and could be used to significantly enhance the feeling of immersion in interactive environments. Also, in a similar way to how physical simulation added new and sometimes unexpected elements to animation, the synthesis of the sounds can also produce effects that were previously unconsidered by the developers. The level of interactivity that synthesised sounds can provide for true dynamic environments can not be matched by current wave-table technologies.

At the stage it is at now however, it is still too complex and unwieldy for use in the mainstream. A number of issues hold back the technology, not least of these being the difficulty in obtaining modal data for specific objects and materials. During the research phase of the project, it was discovered that to obtain accurate modal data experimentally requires the use of sophisticated imaging machinery. This would be a major hindrance to any developer trying to get the necessary data for whatever set of objects they required. Taking the mathematical derivation approach does not seem to provide a desirable

alternative either, with the derivation of equations to calculate modes for more detailed objects being quite an undertaking. The most reasonable solution to this problem is to revise the criteria originally set out and permit the use of pre-processing for data gathering. We outlined before how the work done using finite-element models gave impressive results, but at the cost of extensive processing time. However, seeing as how the modal frequencies do not change during the course of a simulation (assuming the bodies themselves are not subject to changes in geometry) then the only parameters that would need to be determined would be the mode gains. This approach would also allow for the proper implementation of point of impact into the system, as demonstrated in our second test application.

It is unlikely that even with a fully realised synthesis system, that the sample based approach would be abandoned completely. Synthesised sounds are only of advantage to the developers when the components of the simulated world exhibit such varying dynamic properties that sampled sounds will become repetitive and unconvincing after extended use. This however is not likely to be the case for every element of a simulation, so the use of samples for static elements of the environment would be the most appropriate. Also for animations that take place in the background and will only be noticed by the most discerning user, the use of synthesised sounds will be wasted and will only use up much needed processor time.

References

Baraff (1989)

Baraff, D., *Analytical Methods for Dynamic Simulation of non-penetrating Rigid Bodies.* In proceedings of SIGGRAPH '89, Computer Graphics, Boston, 1989.

Baraff (1993)

Baraff, D., *Non-penetrating Rigid Body Simulation*. In proceedings of Eurographics '93 State of the Art Reports, Barcelona, 1993.

Baraff (1995)

Baraff, D., *Interactive Simulation of Solid Rigid Bodies*. Published in IEEE Computer Graphics Applications 15, 1995.

Bastiaans (1980)

Bastiaans, M., *Gabor's Expansion of a Signal into Gaussian Elementary signals*. In proceedings of IEEE Volume 68, 1980.

Bisnovatyi (2000)

Bisnovatyi, I., *Flexible Software Framework for Modal Synthesis*. Proceedings of COST G-6 Conference on Digital Audio Effects, Verona, Italy, 2000.

Boulanger (2000)

Boulanger, R., *The Csound Book: Perspectives in Software Synthesis, Sound Design, Signal Processing,and Programming*. Published by MIT Press, USA, 2000.

Buchanan (1994)

Buchanan, G.R., *Schaum's Outline of Finite Element Analysis*. Published by McGraw-Hill Trade, United Kingdom, 1994.

Cohen et al. (1995)

Cohen, J.D., Lin, M.C., Manocha, D. & Ponamgi, M.K., *ICOLLIDE: An interactive and exact collision detection system for large-scale environments.* In Proceedings of ACM Interactive 3D Graphics Conference, 1995.

Cook (1996)

Cook, P.R., *Synthesis ToolKit in C++ Version*. In proceedings of SIGGRAPH, 1996.

Cook (2002)

Cook, P.R., *Real Sound Synthesis for Interactive Applications*. Published by A K Peters Ltd., 2002.

Demara & Wilson (2002)

Demara, R. & Wilson, J.L., *High Score! The Illustrated History of Electronic Games*. Published by Mc-Graw-Hill/Osbourne, 2002.

Djoharian (1999)

Djoharian, P., *Material Design in Physical Modelling Sound Synthesis*. Proceedings of the 2nd COST G-6 Workshop on Digital Audio Effects, Trondheim, 1999.

Dutoit (1997)

Dutoit, T., *An Introduction to Text-to-speech Synthesis*. Published by Kluwer Academic Publishers, The Netherlands, 1997.

Gabor (1946)

Gabor, D., *Theory of Communication*. Published in Journal of the Institute of Electrical Engineers Part III, 93, 1946.

Hahn (1988)

Hahn, J.K., *Realistic Animation of Rigid Bodies*. In proceedings of SIGGRAPH Computer Graphics Vol. 22, 1988.

Hahn et al. (1995)

Hahn, J. K., Fouad, H., Gritz, L., Lee, J. W., *Integrating Sounds and Motions in Virtual Environments.* Sound for Animation and Virtual Reality, SIGGRAPH 95 Course Notes, 1995.

Hoskinson & Pai (2001)

Hoskinson, R. & Pai, D. K., *Manipulation and Resynthesis with Natural Grains*. In proceedings of International Computer Music Conference, 2001.

Hubbard (1996)

Hubbard, P.M., *Approximating Polyhedra with Spheres for Time-Critical Collision Detection*. In proceedings of ACM Transactions on Graphics, 1996.

Iazzetta & Kon (1999)

Iazzetta, F. & Kon, F., *Downloading Musical Signs*. Retrieved from http://citeseer.ist.psu.edu/101392.html, February 21st, 2003.

Klatzky et al. (2000)

Klatzky, R. L., Pai, D. K. & Krotkov, E. P., *Perception of Material from Contact Sounds*. Appears in Presence, The MIT Press, 2000.

McClellan et al. (1998)

McClellan, J.H., Schafer, R.W. & Yoder, M.A., *DSP First: A Multimedia Approach*. Published by Prentice Hall, USA, 1998.

Moore & Wilhelms (1988)

Moore, M. & Wihelms, J., *Collision Detection and Response for Computer Animation*. In proceedings of SIGGRAPH Computer Graphics Vol. 22, 1988.

O'Brien et al. (2001)

O'Brien, J. F., Cook, P. R. & Essl, G., *Synthesizing Sounds from Physically Based Motion*. In proceedings of ACM SIGGRAPH 2001, 2001.

O'Brien et al. (2002)

O'Brien, J.F., Shen, C. & Gatchalian, C.M., *Synthesising Sounds from Rigid-body Simulations*. From the 2002 ACM SIGGRAPH Symposium on Computer Animation, 2002.

O'Sullivan & Dingliana (2001)

O'Sullivan, C. & Dingliana, J., *Collisions and Perception*. Published in ACM Transactions on Computer Graphics 20, 2001.

Pai et al. (1998)

Pai, D.K., Lang, J., Lloyd, J. & Woodham, R.J., *ACME, A Telerobotic Active Measurement Facility.* Presented at the 1998 IROS Workshop on Robots on the Web.

Pai et al. (2001)

Pai, D.K., van den Doel, K., James, D.L., Lang, J., Lloyd, J.E., Richmond, J.L. & Yau, S.H., *Scanning Physical Interaction Behaviour of 3D Objects.* In proceedings of SIGGRAPH 2001 Conference, 2001.

Pope & Fahlén (1993)

Pope, S. T., & Fahlén, L. E., *The Use of 3-D Audio in a Synthetic Environment: Building an "Aural Renderer" for a Distributed Virtual Reality System*. Swedish Institute for Computer Science, Sweden, 1993.

Preece et al. (2000)

Preece, J., Rogers, Y. & Sharp, H., *Interaction Design: Beyond Human-Computer Interaction*. Published by John Wiley & Sons, USA, 2000.

Roads (1996)

Roads, C., *The Computer Music Tutorial*. Published by The MIT Press, 1996.

Rocchesso et al. (2002)

Rocchesso, D., Rath, M. & Avanzini, F., *Physically-based real-time modelling of contact sounds*. Università degli Studi di Padova, Italy, 2002.

Smith III (1996)

Smith III, J. O., *Physical Modelling SynthesisUpdate.* Original version published in The Computer Music Journal, vol. 20, no. 2, 1996.

Smith III (1999)

Smith III, J. O., *Viewpoints on the History of Digital Synthesis*. Keynote Paper, Proceedings of the International Computer Music Conference, Montreal, 1992.

Takala & Hahn (1992)

Takala, T. & Hahn, J., *Sound Rendering*. Appears in proceedings of SIGGRAPH 1992, Computer Graphics, 1992.

Takala et al. (1993)

Takala, T., Hahn, J., Gritz, L., Geigel, J., Won Lee, J., *Using Physically-Based - Modela and Genetic Algorithms for Functional Composition of Sounds Signals, Synchronized to Animated Motion*. International Computer Music Conference, 1993.

Tzanetakis & Cook (2000)

Tzanetakis, G. & Cook, P., *3D Graphics Tools for Sound Collections*. In Proceedings of the COST G-^ Conference on Digital Audio Effects, Verona, Italy, 2000.

van den Doel & Pai (1996)

van den Doel, K., & Pai, D. K., *The Sounds of Physical Shapes*. University of British Columbia, Vancouver, Canada, 1996.

van den Doel et al. (2001)

Van den Doel, K., Kry, P. G. & Pai, D. K., *FOLEYAUTOMATIC: Physically-based Sound Effects for Interactive Simulation and Animation*. In proceedings of SIGGRAPH 2001 Conference, 2001.

van den Doel & Pai (2003)

van den Doel, K., & Pai, D. K., *Modal Synthesis for Resonating Objects.* Retrieved June 7, 2003, from http://www.cs.ubc.ca/~kvdoel/acel/workshop/doel/bio.html

Watt & Policarpo (2003)

Watt, A. & Policarpo, F., *3D Games – Animation and Advanced Real-time Rendering: Volume 1*. Published by Addison Wesley, 2003.

Witkin et al (1990)

Witkin, A., Gleicher, M., & Welch W., *Interactive Dynamics*. In proceedings of Computer Graphics (1990 Symposium on Interactive 3D Graphics), 1990.

Young & Freedman (1996)

Young, H.D. & Freedman, R.A., *University Physics*. Published by Addison Wesley Publishing Company, Inc., 1996.

Zonst (1995)

Zonst, A.E., *Understanding the FFT: A Tutorial on the Algorithm & Software for Laymen, Students, Technicians & Working Engineers*. Published by Citrus Press, USA, 1995.