Contact Sound Synthesis in Real-time Applications

CPU-based Multicore Synthesis in the Frequency Domain

Robin Lindh Nilsson

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden
This thesis is submitted to the Department of Creative Technologies at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Game and Software Engineering. The thesis is equivalent to 20 weeks of full-time studies.

Contact Information:
Author: Robin Lindh Nilsson
E-mail: robinerd@gmail.com

University advisor:
Dr. Martin Fredriksson
Dept. Creative Technologies

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone    : +46 455 38 50 00
Fax      : +46 455 38 50 57
Abstract

**Context.** Synthesizing sounds which occur when physically-simulated objects collide in a virtual environment can give more dynamic and realistic sounds compared to pre-recorded sound effects. This real-time computation of sound samples can be computationally intense.

**Objectives.** In this study we investigate a synthesis algorithm operating in the frequency domain, previously shown to be more efficient than time domain synthesis, and propose a further optimization using multi-threading on the CPU.

**Methods.** The multi-threaded synthesis algorithm was designed and implemented as part of a game being developed by Axolot Games. Measurements were done in three stress-testing cases to investigate how multi-threading improved the synthesis performance.

**Results.** Compared to our single-threaded approach, the synthesis speed was improved by 80% when using 8 threads, running on an i7 processor with hyper-threading enabled.

**Conclusions.** We conclude that synthesis of contact sounds is viable for games and similar real-time applications, when using the investigated optimization. 140000 mode shapes were synthesized 30% faster than real-time, and this is arguably much more than a user can distinguish.

**Keywords:** Sound synthesis, parallel computing, auditory feedback
Acknowledgments

My sincerest thanks go to my supervisors Dr. Martin Fredriksson and Dr. Johan Hagelbäck for all of their useful feedback, guiding my thesis work with great passion.

I am also very thankful for the fantastic opportunity to do my thesis in cooperation with Axolot Games where Kacper Antonius and Pontus Holmbom motivated me a lot and helped me to utilize all the benefits of their game code. Without them this thesis would not have been possible.

The many discussions with Björn Wennerström and Alexander Brodén greatly helped to find inspiration, and really contributed well towards the final result.

Finally, I wish to express my honest gratitude to my parents and my girlfriend for their constant motivation, always supporting me during my studies.
# Contents

Abstract

List of Figures

1 Introduction
   1.1 Motivation .............................................. 1
   1.2 Related Work .......................................... 2
   1.3 Problem Statement ..................................... 3
   1.4 Aim ...................................................... 3
   1.5 Approach ............................................... 4
   1.6 Thesis Outline ........................................ 4

2 Background

3 Implementation
   3.1 Frequency Domain Synthesis .......................... 9
   3.2 Real-time Synthesis Pipeline ......................... 10
   3.3 Physics Integration .................................... 12
   3.4 Singlethreaded Solution ............................... 13
       3.4.1 Architecture Overview .......................... 13
       3.4.2 Algorithm Outline .............................. 14
   3.5 Multi-threaded Adaption ............................... 15

4 Experiment Method
   4.1 Experiment Setup ....................................... 17
   4.2 Measuring the Performance ............................ 19
   4.3 Execution ............................................... 20
   4.4 Method Evaluation ..................................... 20

5 Results
   5.1 Peak Scenario .......................................... 23
   5.2 Continuous Scenario .................................. 24
   5.3 Big Objects Scenario .................................. 25
5.4 Synthesis Performance

6 Analysis

7 Conclusions and Future Work

References

A Glossary

B FMOD Synthesis Setup
List of Figures

2.1 The signal \( s(t) = 0.7 \sin(t) + 0.3 \sin(2t) + 0.2 \sin(4t) \), represented in the time and frequency domains. ......................... 7

3.1 An illustration of how audio sources connect modes to a rigid body. 12
3.2 The main synthesis classes and their relations. ....................... 13
3.3 The different threads and their interactions. ......................... 14

4.1 A screenshot from the game, with sound sources visualized by red circles. ......................................................... 19

5.1 Peak Scenario: Number of modes ................................. 23
5.2 Peak Scenario: Execution time factor ............................ 23
5.3 Continuous Scenario: Number of modes ......................... 24
5.4 Continuous Scenario: Execution time factor ..................... 24
5.5 Big Objects Scenario: Number of modes ....................... 25
5.6 Big Objects Scenario: Execution time factor .................... 25
5.7 Execution time factor depending on mode count ................ 26
Chapter 1

Introduction

1.1 Motivation

In virtual environments (VE), sound effects are commonly used to signal or emphasize various events to the user, and are traditionally generated through playback of recorded sounds. However, the sounds produced when physically simulated objects interact (“contact sounds”), pose a particular challenge to accurately represent with prerecorded sounds since they depend on many physical parameters. An emerging alternative approach is to generate contact sounds with physically-based synthesis methods. This presents many new possibilities for realism and variety, but is still rarely seen in commercial VE applications. To address this, our study aims to find a portable high-performance synthesis method for multi-core CPUs, and ideally make sound synthesis more viable for commercial VE applications. The study will be supervised by Axolot Games\(^1\), which has expressed great interest in this technology, particularly due to the complex matter of representing contact sounds using prerecorded samples.

When synthesizing sound in real-time, it is critical that sound samples can be generated at audio playback rate\(^2\) or faster. If not, the lack of available sound information may cause undesirable sound artifacts such as stutter or silence. Additive modal sound synthesis is based on the fact that any sound can be represented using a sum of sinusoidal waves. By improving the performance so that more modes can be synthesized in real-time, any particular sound can be more accurately represented, which in turn can increase the realism of existing techniques for resembling real materials\(^3\).

\(^1\) http://www.axolotgames.com
\(^2\) Playback rate refers to the number of sound samples per second played by the audio device.
\(^3\) Such techniques were discussed by Ren et al. (2013) and O’Brien et al. (2002), among others.
Chapter 1. Introduction

While synthesis has traditionally been performed in the time-domain (TD) where a signal is represented as a function over time, previous research by Bonneel et al. (2008) describes an efficient method of synthesizing sounds in the frequency domain (FD). In the FD a signal is instead represented through coefficients telling how much of each frequency interval is present in the signal, and thanks to approximating these coefficients with high performance their study was able to reach a higher synthesis speed. A more detailed description of the time and frequency domains is given in chapter 2 Background. This method of FD synthesis was found very promising and was used as the base in this thesis, extended with further optimizations for even higher performance on multi-core CPUs.

1.2 Related Work

Sound synthesis has been the topic of many previous studies, such as the ones by van den Doel et al. (2001), O’Brien et al. (2002), Van den Doel and Pai (2003), and James et al. (2006). All of these perform the synthesis in the time domain.

An early optimization was demonstrated by van den Doel et al. (2004), which increased the synthesis performance in large scenes by only rendering the modes judged to be audible to a user. This technique has inspired many more recent studies, and is a concept which we also used in this thesis.

Raghuvanshi and Lin (2006) obtained convincing results using a CPU-based approach and a number of novel optimizations; 100 sound generating objects were simulated twice as fast as real-time CD-quality rates\textsuperscript{4}, running on a single-core Pentium 4 processor. Bonneel et al. (2008) further increased the performance by a factor of 5-8 by performing synthesis in the frequency-domain instead of the time-domain. The key part of their study was an equation to efficiently approximate the frequency-domain coefficients of the sound, which proved to be faster than directly computing the signal in the time domain, when having many simultaneous sounds. The described FD synthesis was chosen as the base of this thesis due to its high performance.

Although the CPU based studies mentioned above achieved impressive results, their implementations were entirely sequential, and thus cannot fully utilize multi-core CPUs. Cahill (2009) showed that parallelizing previous methods of time-domain synthesis gave a significant speed-up, but due to a variety of problems encountered they did not evaluate all attempted optimizations. Although not fully reaching their initial aim, their research reveals many considerable obstacles, and the range of possible improvements motivates the further research performed in this thesis.

\textsuperscript{4} A common audio playback rate is 44100 Hz, here referred to as CD quality.
Chapter 1. Introduction

1.3 Problem Statement

This thesis investigates the possibilities of increasing the performance when synthesizing contact sounds in a CPU-based implementation. The aim was to find an approach capable of real-time execution with a high number of sounding modes. In this thesis, modes refer to the sinusoidal waves which are added together in the synthesis of contact sounds. Modal synthesis is further described in 2 Background.

This thesis aims to answer the following research questions:

1. How much can multi-threading improve the performance of the frequency domain sound synthesis? Performance refers to the number of modes that can be synthesized in real-time.

2. Is CPU-based multi-threaded sound synthesis viable for synthesizing contact sounds in a VE application?

Real-time in this case means that sound samples are synthesized at least at the rate required by the sound playback device. A more detailed definition is given in 4.2 Measuring the Performance.

1.4 Aim

Previous studies have achieved real-time speed by utilizing the great parallelism of modern GPUs. However, a CPU-based solution with enough performance could help making sound synthesis easier to implement and maintain in VE applications, without interfering with the rendering pipeline.

Overall, although several previous studies address CPU-based contact sound synthesis, and performance improvements have been suggested, no approach has successfully combined all of these into a working application. This study will focus on parallelizing frequency-domain synthesis, to measure and ideally improve the performance on modern CPUs in relation to previous attempts.

A high-performing synthesis implementation may also leave enough CPU time to use sound propagation effects for even more realism. While this thesis does not address sound propagation, we intend to make such a combination more viable by improving the synthesis performance.
1.5 Approach

To answer the research questions an implementation was done of the frequency domain (FD) synthesis by Bonneel et al. (2008), as part of a game in development by Axolot Games who supervised this thesis. We started with implementing the FD synthesis algorithm in a simplified scenario, and then extended this with a connection to the game physics. Some ideas were based on other studies as well since little information about the FD implementation could be found.

When the FD synthesis was finished and working as expected, it was modified to use multiple threads. Our approach to multi-threading was partly inspired by ideas from Cahill (2009), but adapted to be used in FD synthesis. Also, we divide the work-load differently because of implementation details.

Finally, measurements were done to find the performance in relation to the number of modes, and see how this relation was improved by using more threads.

1.6 Thesis Outline

The rest of this thesis is divided into several chapters describing our multi-threaded implementation of frequency domain synthesis of contact sounds. The thesis only covers CPU-based synthesis, and emphasizes how is can be used in an actual game, with connection to the physics engine to trigger contact sounds. Note that the aspect of resembling realistic materials has been discussed in several previous studies, and is entirely left out in our implementation. Instead, this thesis focuses on increasing the performance, which in turn makes it possible to better represent realistic materials by using more modes.

A general introduction to sound synthesis is given in chapter 2 Background. In the next chapter, 3 Implementation, frequency domain synthesis is thoroughly explained followed by technical details about how it was used in an actual game to synthesize contact sounds. This includes communications with the physics engine as well as a high-level description of the architecture and the algorithm flow. In chapter 4 Experiment Method we present all details regarding the performance measurements, such as the test cases, the hardware used and the measurement procedure itself.

All measurement results are presented in chapter 5 Results, followed by further discussions and comparisons with previous studies in chapter 6 Analysis. Finally, our contribution is put into a wider perspective in chapter 7 Conclusions and Future Work.

There are also two appendices; A Glossary with explanations on some of the
common terminology used, and \textit{FMOD Synthesis Setup} in which a basic code sample can be found.
Sound Synthesis

Sound can often be the result of playing samples which have been previously recorded and processed. Sound synthesis on the other hand is a technique where a computer generates the sound signal according to an algorithm. There are many different ways of doing this, but they all result in a sound wave being calculated based on some set of parameters. The specific technique used in this thesis is sinusoidal additive synthesis which works by adding a large number of computed sinusoidal waves together to approximate an arbitrary sound. In this topic the individual waves are called modes. A great introduction to the field of sound synthesis is given in the literature by Cook (2002).

A mode consists of a certain angular frequency ($\omega$), phase offset ($\phi$) and volume ($V$), and is also commonly represented on a complex form. In this thesis the synthesis is used to generate sounds when objects collide (“contact sounds”), so each mode also has a decay rate ($\alpha$) specified.

Time Domain (TD)

Audio modes are often represented with a function giving the signal $s$ as a function of time ($t$), as in the upper graph in Fig. 2.1. This is referred to as the time domain (TD). The synthesis of contact sounds in the TD can be described by Eq. 2.1, where $N$ is the number of modes and $\alpha < 0$.

$$s(t) = \sum_{n=1}^{N} V_n e^{\alpha_n t} \sin(\omega_n t + \phi_n)$$  \hspace{1cm} (2.1)
Figure 2.1: The signal $s(t) = 0.7 \sin(t) + 0.3 \sin(2t) + 0.2 \sin(4t)$, represented in the time and frequency domains.
Chapter 2. Background

Frequency domain (FD)

In contrast to the time domain, sinusoidal waves can also be represented as a complex function which describes the magnitude as a function of frequency. This is called the frequency domain (FD) and can basically be thought of as how much the wave contains of each frequency, illustrated in the lower graph in Fig. 2.1. A single sine wave would have one spike in its FD function, corresponding to its frequency. Similarly a waveform consisting of many waves added together would have many spikes according to each frequency, scaled by their magnitudes. Note that each frequency corresponds to a complex number, where the real and imaginary coefficients correspond to the cosine and sine parts of the wave, respectively. For more details about complex sinusoidal waves, and complex numbers in general, the reader is referred to mathematics literature.

Fourier Transforms

The Fourier transform is a mathematical transformation to convert a signal from the time domain into the frequency domain. This can be used on any periodic signal to find which frequencies it consists of, and the magnitude of each. To go the other way around the inverse Fourier transform is instead used.

Often, the discrete Fourier transform (DFT) is used instead, which approximates a continuous signal into discrete intervals. In the time domain, this means that the signal is divided into fixed sound samples, commonly at a rate of 44100 samples per second in digital audio playback. In the frequency domain, the frequency spectrum is divided into ranges, referred to as bins. The number of bins depends on the resolution of the Fourier transform; more bins means higher precision but slower calculation speed. In this thesis 1024 bins are used.

When the DFT is implemented in a computer application an optimized implementation is often used, called the fast Fourier transform (FFT). This gives the same result as the mathematical definition but with a lower execution time. Similarly there is also the inverse fast Fourier transform (IFFT).
Chapter 3

Implementation

3.1 Frequency Domain Synthesis

To find out whether sound synthesis for contact sounds can be viable for real-time applications such as games, a choice was made to perform the modal synthesis in the frequency domain (FD), as described by Bonneel et al. (2008), since this approach was up to eight times faster than time domain (TD) synthesis according to their studies. The implementation in this thesis was overall based on their method (henceforth referred to as “the original method”) but may differ in some parts since not much detail about their implementation could be found. Also, one of their optimizations for contact sounds, called temporal scheduling, was left out. It improves performance when many sounds are playing by slightly delaying some of the modes to mitigate the spike. This optimization was not implemented in this thesis, meaning that there is still potential for improvement over the approach presented in this thesis. Below, the core synthesis algorithm is described in general terms.

In TD synthesis, each sound sample is computed individually, giving the amplitude as a function of time. The amplitudes are computed for each sounding mode, and added together to obtain the total amplitude to output. For a general explanation of sound synthesis, see 2 Background.

In FD synthesis, samples are not computed individually. Instead, the output signal is divided into windows of $C$ samples, and each window is computed as a whole. In our implementation $C = 1024$ was used, based directly on the window size in the original method. The reader is assumed to be familiar with complex numbers and discrete Fourier transforms, which are central concepts in this method. Basic explanations of these are given in 2 Background.

To compute one window of the output signal, the Fourier coefficients of all modes are first calculated. The original method presents a formula for approximating
these (Eq. 3.1), which is much faster than computing an actual Fourier transform. In our implementation we derived this into Eq. 3.2 using Euler’s formula. A variable \( V \) was introduced to control the overall volume of the mode. Also, the exponential decay was rewritten on a recursive form to be calculated on a per-frame basis, allowing the exponential \( e^{\alpha \Delta t} \) to be precomputed for better performance. It is essentially still the same equation, but shows better how it was actually implemented in our approach. Each mode is approximated, which gives a vector of \( C \) complex numbers, corresponding to the \( C \) frequency bins of the discrete Fourier transform \( \mathcal{F}_\lambda \). All mode coefficients are added, per bin, to get the FD representation of the total signal, in the current window. Finally an inverse Fourier transform (IFFT) is applied to convert the signal into \( C \) TD samples.

\[
s(\lambda) \approx \frac{1}{2} e^{\alpha t_0} c_0 i \left( e^{-i \omega t_0} \mathcal{F}_\lambda (H)(\lambda + \omega) - e^{i \omega t_0} \mathcal{F}_\lambda (H)(\lambda - \omega) \right)
\]

where \( c_0 = \frac{e^{-\alpha t_0} - e^{-\alpha (t_0 + \Delta t)}}{\alpha \Delta t} \)

\[
s(\lambda) \approx \frac{1}{2} VA_t c_0 (k_{\text{sub}} \mathcal{F}_\lambda (H)(\lambda + \omega) - k_{\text{add}} \mathcal{F}_\lambda (H)(\lambda - \omega))
\]

\[
k_{\text{add}} = \cos(t_0 \omega) - i \sin(t_0 \omega)
k_{\text{sub}} = \cos(t_0 \omega) + i \sin(t_0 \omega)
A_t = A_{t-\Delta t} \cdot e^{\alpha \Delta t}
\]

### 3.2 Real-time Synthesis Pipeline

To integrate the FD synthesis in a real-time application, a “synthesis pipeline” was designed, and implemented in C++. In this, windowing is done on the fly to synthesize a continuous audio stream. Each window is computed according to the process described above and the samples are written to the audio buffer. For every window, all Fourier coefficients must be re-estimated since the starting time \( t_0 \) used in Eq. 3.2 is constantly increasing.

The first challenge was to setup the audio framework in a way that enables custom program code to generate audio samples in real-time, as opposed to playback of sound files. For this, a choice was made to use FMOD\(^1\); it is well documented and easy to use for this task, and as of the time of writing also free to use in commercial projects with budgets less than 100000 USD.

\( ^1 \) http://www.fmod.com
Using FMOD, a sound channel was setup to compute sound samples using a custom callback function. This will be referred to as the "synthesis channel". FMOD then handles the playback and repeatedly executes the callback function to fill the internal sound buffer. Using FMOD for this setup avoided a lot of involved sound programming, leaving more time for implementing the actual synthesis. See appendix B FMOD Synthesis Setup for a basic code example.

A great strength of the proposed pipeline is that the synthesis channel can be treated just like the channels for recorded sound files, which for example means that all tools for digital sound processing in FMOD can be used to enhance the synthesized output as well.

Some restrictions are implied by this pipeline due to the fact that all synthesized sounds are mixed together and output to the same sound channel. Although it is theoretically possible to synthesize each sound source in a separate sound channel, it may reduce the performance. In the FD synthesis method, all sounding modes are added in the frequency domain, followed by only one single IFFT calculation. If modes were to be synthesized separately, the IFFT would be performed for every mode. Due to this any effects applied on the synthesis channel will affect all synthesized sounds.

Since all synthesized sounds are played by the same channel, FMOD's built-in 3D sound cannot be utilized for the synthesis. Instead, the channel was set to play as a 2D sound, and a custom implementation of attenuation and stereo panning was done. Controlling the volumes of each mode separately must be done in the frequency domain, by scaling the volume $V$ of each mode in Eq. 3.2. Other effects may be implemented in a similar way, but since the individual modes can only be modified in the FD, only FD effects can work on a per-mode basis. TD effects, as stated, can still be implemented to operate on the mixed result of all modes.

Once FMOD had been setup with a synthesis callback, the callback was implemented with the FD synthesis algorithm previously explained. For the IFFT calculations, Kiss FFT was used\(^2\). It may not be the fastest option available, but seemed to be easy to use. As explained by Bonneel et al. (2008) the IFFT calculation is negligible even using an unoptimized implementation, so ease of use was considered more important in this case. Kiss FFT also comes with functionality for complex numbers. In retrospective, some details were lacking documentation, and there may be more suitable choices. Still, the choice of IFFT implementation is not critical for the overall algorithm.

\(^2\)https://www.ceemple.com/utility-libraries/math/kiss-fft/
3.3 Physics Integration

Synthesizing contact sounds mainly involved three steps: detecting impacts in the simulated physics, determining which modes should be triggered, and playing them in the synthesis pipeline with the correct volume. All modes are initially silent, but upon impact the amplitude is increased causing it to play. The amplitude of each mode is decaying exponentially over time with its decay rate $\alpha$.

![Figure 3.1: An illustration of how audio sources connect modes to a rigid body.](image)

To represent the sounds in relation to the virtual world the concept of sound sources was introduced (Fig. 3.1). A sound source ($S_j$) is a collection of modes ($M_n$), attached to a rigid body with an offset $\vec{P}_j$ from the body’s origin. By changing the number of modes and their parameters (the “mode configuration”) different sounds can be obtained, to mimic different materials. Multiple sound sources can be attached to different parts of the same body, with different mode configurations, to get different sounds depending on the impact location, for example with regards to the object’s shape.

When an impact is reported by the physics simulation a contact sound is generated. The sound source closest to the impact location is selected, and all of its modes have their amplitudes increased by $\Delta A$ defined by:

$$\Delta A = k \cdot \left( (v_1 + \vec{P}_1 \times \vec{\omega}_1) - (v_2 + \vec{P}_2 \times \vec{\omega}_2) \right) \cdot \vec{n} \quad (3.3)$$

This is simply the relative speed of the colliding points on each object, projected along the collision normal $\vec{n}$, and scaled by a factor $k$. The value of $k$ needs to be balanced in relation to the scale of the VE. The amplitude increase is distributed over a short time span to prevent a notable break in the sound wave. Both of the colliding objects trigger contact sounds in this way, independently of each other. Note that it is also possible to generate contact sounds using more than
one sound source as described by Zheng and James (2011) which may improve the realism at the expense of lowered synthesis speed. In this thesis, only the closest sound source generates sound.

Since our implementation was done in a game being developed by a company, the synthesis was integrated with the physics engine already used in their application, which is Bullet Physics\(^3\). No particular problems were encountered, and no other physics engines were investigated.

### 3.4 Singlethreaded Solution

#### 3.4.1 Architecture Overview

In the synthesis implementation a number of classes were implemented to model the synthesis algorithm previously described. A basic class diagram with the relations between these can be seen in Fig. 3.2.

![Figure 3.2: The main synthesis classes and their relations.](http://bulletphysics.org)

Rigid bodies are created and managed by the physics engine. Each time a rigid body is added, removed or changed, the synthesis container *AudioSynth* is notified, and makes sure to always have exactly one *SynthObject* per rigid body. Each *SynthObject* generates a number of *SynthSources* based on the shape of the rigid body, and if the shape of the body changes the generation is repeated to match the new shape. The *SynthSources* create and manage their *SynthModes*.

---

\(^3\) [http://bulletphysics.org](http://bulletphysics.org)
3.4.2 Algorithm Outline

The main steps of the algorithm are shown in Fig. 3.3. Each system represents a separate thread in our implementation, and some locking mechanisms were used to guarantee thread safety. Three main parts are illustrated:

1. keeping the synthesis representation up to date when rigid bodies are added, removed or changed,
2. increasing the amplitude of the relevant modes upon collisions, and
3. computing the synthesized audio based on all currently playing modes, and writing it to the buffer.

Figure 3.3: The different threads and their interactions.
3.5 Multi-threaded Adaption

Out of the whole synthesis algorithm, calculating and adding the approximated FD coefficients stands for almost all of the execution time. This step, labeled "Calculate FD coefficients" in Fig. 3.3, is the part which we modified to run with multiple threads. The other parts of the synthesis may possibly be multi-threaded as well for a slight performance improvement, but since those parts are only triggered when objects collide or change shape they are less critical to improve, and were not investigated in this thesis. Further, these other parts heavily depend on the physics engine and could thus be more difficult to successfully parallelize.

The main idea of the multi-threading was that there are many modes, and their FD coefficients can be computed independently of each other, making the task very suitable for a parallel approach. The first attempt was done by creating a number of threads for each block of 1024 sound samples, and splitting the modes equally between the threads. This approach proved slower than the single-threaded implementation which may have been due to the high overhead of creating threads.

A new approach was taken in which a pool of worker threads is created when the application starts. This thread pool can be reused during the whole execution to avoid the overhead of thread creation. The sounding modes are divided into small partitions of data such that the number of partitions is much higher than the number of threads. These partitions are shared between the worker threads, and each thread is directly assigned with a new partition as soon as the previous one has been calculated. This approach not only has less overhead, but also improves the load balancing since the workload is continuously shared between all worker threads. The exact partition size may be tweaked in order to optimize the trade-off between load balancing and partition assignment overhead. In our implementation we got satisfactory results by choosing the number of partitions to be three times more than the thread count, but no in-depth study was performed since the difference seemed to be very small in relation to the actual work performed.

To implement the threading described above, boost::threadpool\(^4\) was used which is an extension to the Boost C++ library\(^5\). This was chosen because of its ease of use and thorough documentation, and was particularly suitable to integrate with the rest of the application. Though, the multi-threading approach does not dependent on this particular library; any implementation of a thread pool could be equally suitable.

The main challenge with the multi-threading was that accessing the list of all

\(^4\) [http://threadpool.sourceforge.net](http://threadpool.sourceforge.net)

\(^5\) [http://www.boost.org](http://www.boost.org)
sounding modes must be easy and efficient. Since this list of sounding modes is affected by both the synthesis thread, the physics thread and the thread where object modifications are detected, a locking mechanism was necessary to synchronize the list between threads. The most efficient solution we found was that instead of letting each sound source manage its own sounding modes, each mode adds or removes itself to a global list of sounding modes. When an impact is reported and modes start playing, they add themselves to this list. Similarly, a mode removes itself when its volume goes below some epsilon limit. In this way the locking between threads is minimized to achieve high performance gains with the multi-threading, and a list of only the sounding modes is kept up to date so that only the sounding modes need to be calculated. This idea of keeping a list of only the sounding modes was based on the original method by Bonneel et al. (2008) but adapted to match our multi-threaded approach and class hierarchy.
Chapter 4

Experiment Method

4.1 Experiment Setup

To investigate whether multi-threaded FD synthesis can be viable for generating contact sounds in a VE application, the proposed approach was implemented as described in chapter 3 Implementation. The performance was then measured to see how many modes could be synthesized in real-time, and how much this number improved when using more threads.

Three different test scenarios were designed in order to stress test the synthesis in different ways based on probable bottlenecks. Sound sources were automatically assigned to each object, distributed over its surface, meaning that bigger objects have more sound sources attached. For each sound source 3000 modes were assigned, with their mode parameters randomized within certain limits. The limits were designed to generate a harmonic sound which resembles a metallic clank. Although not part of this thesis, it should be noted that any material could be approximated in this way by altering the mode parameters accordingly.

1. Peak Scenario
The first scenario consists of a large number of rigid bodies falling from the sky, all colliding with the flat ground in a short time span. Only one sound source was attached to each object to allow for more objects. This was seen as the ultimate stress-test on the real-time synthesis; a lot of collisions to be handled, and a lot of modes playing at the same time.

2. Continuous Scenario
In the second scenario a more continuous flow of collisions was constructed using motors and rotational joints connected in a way that a large amount of objects were rotating and colliding repeatedly. This setup kept the number of sounding modes fairly constant, at the edge of real-time performance, to study potential issues which could arise when being close to this limit over some period of time.
One such issue could potentially be if the synthesis gets slower over time by some overhead.

3. Big Objects Scenario
The final test case had three big ellipsoid objects made up of around 300 smaller cubes. Each small cube had a sound source attached. Since only the sound source closest to the impact is triggered to emit sounds, this scene does not cause a large amounts of modes to play. Rather, the aim was to test for any potential overhead caused due to dealing with many sound sources when triggering the sound. One theory was that looping over all sound sources in order to find the closest one could slow down the synthesis.

<table>
<thead>
<tr>
<th>Computer Model:</th>
<th>Asus X53S</th>
</tr>
</thead>
<tbody>
<tr>
<td>Motherboard Model:</td>
<td>K53SC</td>
</tr>
<tr>
<td>Operating System:</td>
<td>Windows 7 Home Premium 64-bit</td>
</tr>
<tr>
<td>CPU:</td>
<td>Intel Core i7-2670QM, 2.20 GHz</td>
</tr>
<tr>
<td>RAM:</td>
<td>8192 MB</td>
</tr>
<tr>
<td>GPU:</td>
<td>Nvidia GeForce GT 630M, 2GB</td>
</tr>
<tr>
<td>Display Mode:</td>
<td>1366 x 768, 60Hz</td>
</tr>
<tr>
<td>DirectX version:</td>
<td>DirectX 11</td>
</tr>
</tbody>
</table>

Table 4.1: The hardware used for executing the experiments.

All experiments were performed on an Asus X53S laptop with the hardware specifications seen in Table 4.1. The operating system was Windows 7 Home Premium, and no other programs were running while measuring except for operating system processes and drivers. The antivirus software was turned off and the network connection disabled, to avoid any undesired computational spikes. The computer was set to the default high-performance mode with the charger connected during the whole experiment. Although the laptop also has an integrated Intel HD graphics chip, the application was set to only use the dedicated graphics card listed in the hardware table.

In the experiments hyper-threading was enabled, which causes each hardware core to expand into two logical cores. Hyper-threading has been seen to increase the performance, especially in applications where the number of threads is higher than the number of cores. Measurements presented by Mattwandel et al. (2009) encouraged this, so we decided to use hyper-threading in the experiments to better resemble a real-world scenario.

All performance measurements were done in a computer game being developed by Axolot Games. Specifically, the studied part was the audio synthesis implementation which has been described in chapter 3 Implementation. A screenshot from
the game is presented in Fig. 4.1, taken from the continuous scenario. The red circles were added to visualize the sound sources, with the radius being dependent on the sound volume.

Figure 4.1: A screenshot from the game, with sound sources visualized by red circles.

4.2 Measuring the Performance

The main aim of the experiments was to investigate how many modes could be synthesized in real-time, in relation to the number of threads. Real-time was defined as “being able to synthesize sound samples faster than the audio playback rate”. As stated, the synthesis works by continuously computing chunks of 1024 sound samples on demand by the sound playback device, which are then written to the sound buffer. This chunk of 1024 samples corresponds to $1024/44100 \approx 0.023$ seconds of sound data at the playback rate of 44100 Hz, and to prevent artifacts like noise or sound stutter each chunk must be generated in less than that time. In practice a sound buffer is used to mitigate much of the noise caused by temporary computational spikes, but in any case the average execution time must still be faster than this *real-time* limit. In the experiments we wanted to avoid any dependency on a specific buffer length, so instead each individual chunk was measured in relation to this limit. The *execution time factor* was defined as the time needed to synthesize the chunk, divided by the duration of
the generated sound. A chunk was defined to achieve real-time performance if it had an execution time factor less than 1.

Measuring the execution time of synthesizing a chunk was implemented using the `boost::chrono::high_resolution_clock` function in the C++ standard library. By calling this, high precision time stamps were obtained both before and after the synthesis code, which were then used to calculate the elapsed time. The duration of the generated sound was calculated by dividing the number of samples generated by the playback rate of 44100 Hz. All measurements were written to an output file, for each generated chunk, and later processed to produce the results.

4.3 Execution

For each test scenario the number of threads was varied, using 1, 2, 4, 8, 12 and 16 threads. The upper limit was chosen since the CPU used has 4 hardware cores, expanded to 8 logical cores by the hyper threading. In a pilot study we found that the top performance was around 8 threads, and at 12 threads and higher the performance did not increase further, so higher thread counts were omitted in the final experiment.

With three scenarios and six thread configurations there were in total 18 different test cases. Each test case was executed for six seconds while continuously monitoring the performance. Although the starting setup was always identical for each scene, the physics simulation itself is not entirely deterministic. Therefore, to increase the reliability of the results and estimate the error spread, each test case was repeated three times.

The whole experiment procedure was entirely automated by a separate program to not depend on any user interactions. Further, all experiments were done consecutively at one time, with the program being restarted each time to prevent any potential memory leaks or problems to affect the following test cases. In total, these steps helped to assure that all measured data were collected under the same circumstances and can safely be compared.

4.4 Method Evaluation

In the experiments, hyper-threading was used as well as the automatic thread scheduling in the operating system. An alternative would be to turn off the hyper-threading and assign each thread to a fixed core. Although this alternate approach may achieve more stable results by avoiding these external factors, the performance was likely higher in our setup. As shown by Mattwandel et al. (2009)
Chapter 4. Experiment Method

the hyper-threading improves the performance of programs with many threads, and the thread scheduling helps to improve the load balancing between cores. Still, it is important to clarify the differences between these different approaches, and to gain more knowledge it would be necessary to try both.

Measuring execution time precisely with a minimal impact on the application is a notorious challenge. The actual execution time of the synthesis was measured using two calls to `boost::chrono::high_resolution_clock`, as previously described. According to a pilot study we performed using the profiling tool in Visual Studio 2012, this function call had a very low overhead in relation to how often it was executed in our experiments, and gave a reliable precision in the magnitude of nano seconds. Furthermore, this overhead was constant and affected all test cases in the same way. We conclude that this overhead did not impact the reliability of this study.

Doing file output in the experiments could arguably affect the measured time, but was necessary to save the results. To minimize this impact, the file output stream was kept open throughout the whole execution. It should also be noted that the rest of the application barely uses the hard drive during the execution. An optional approach would have been to store all measurements in the memory and writing the results after finishing the experiment, but this could also affect the results since the memory usage would then increase over time. In retrospect the latter approach would have been preferable, but since this possible overhead is the same in all test cases it had likely no significant impact on the final results.

When performing the experiments each sound source originally had 200 modes associated since Raghuvanshi and Lin (2006) pointed out that a few hundred modes is enough to represent most materials. However, using this setup the high number of physics objects required to push the synthesis performance down below real-time could not be simulated in real-time by the physics engine. It is not clear why the physics engine met this restriction as many existing applications can simulate much more objects, but it was left outside of this thesis to optimize the physics code. Instead, the issue was dealt with by increasing the number of modes per sound source to 3000. This may have caused some of the computational spikes to be slightly higher since a lower number with more objects could have distributed the impacts better over time. Though, since we’re only interested in the performance in relation to the number of playing modes, and the amount of playing modes is still correctly measured, the impact on the results should be negligible.
Chapter 5

Results

One section is given below to illustrate the sounds generated in each scenario. Each example consists of one graph showing how the number of modes changes during the execution, and a second graph showing the execution time factor over the same duration. These graphs are average values out of three runs. It can be stated that the number of modes is rather stable between executions, as it should be, and that the execution time factor is in general lower when more threads are used.

Approximately one second is initially silent in all scenarios since the synthesis starts before objects are loaded by the physics engine. Also, due to small variations in this loading time, all data series have slightly different offsets along the time axis. Some variations were also caused by the physics engine not being entirely deterministic.
Chapter 5. Results

5.1 Peak Scenario

As seen in Fig. 5.1 the peak scenario quickly reaches up to a large number of modes with peak values ranging from 134000 to 138000 modes. The sudden peak builds up when the large number of cubes falling from the sky collide with the ground in a very short time span. After the main impact, several smaller bounces follow, causing some subtle spikes during the decaying part of the graph. In Fig. 5.2 the execution time factor is seen to decrease using more threads, with the peaks being at 1.46, 1.10, 0.89, 0.76 and 0.77, in order of increasing thread counts.

![Figure 5.1: Peak Scenario: Number of modes](image1)

![Figure 5.2: Peak Scenario: Execution time factor](image2)
5.2 Continuous Scenario

The modes in the continuous scenario, presented in Fig. 5.3, start from zero and gradually build up to a stable level as the motor objects start moving. In the interval from 2 – 6 seconds all objects are generating sounds, but since the collisions happen at slightly irregular intervals there are still some variations in the mode level. In this interval, 90% of the execution time factor samples (Fig. 5.4) are in the range 0.66 – 0.96 when using a single thread, but with some peaks still above the real-time limit. Using 8 threads, this range was reduced to 0.40 – 0.54, with all samples being notably faster than real-time.

Figure 5.3: Continuous Scenario: Number of modes

Figure 5.4: Continuous Scenario: Execution time factor
Chapter 5. Results

5.3 Big Objects Scenario

In the big objects scenario much fewer modes are triggered. Although each big object had around 300 sound sources, only a few of these were triggered in the collisions, resulting in few modes (Fig. 5.5). It is also clear from Fig. 5.6 that the large number of non-sounding modes did not cause any notable impact on the execution time factor. While the single-threaded test case was slightly slower than the multi-threaded ones, no significant improvement was observed when using more than two threads.

Figure 5.5: Big Objects Scenario: Number of modes

Figure 5.6: Big Objects Scenario: Execution time factor
5.4 Synthesis Performance

To get a clear view on how the number of sounding modes affects the synthesis speed, all data samples in the three executions of the peak scenario were examined. All samples were categorized into intervals depending on the number of sounding modes, with the intervals being $0 - 10000$, $10000 - 20000$ and so on. For each interval an average of all modes was calculated, per thread configuration. The results are presented in Fig. 5.7, with the x-axis being labeled according to the upper limit of the ranges. For example, 20k corresponds to samples with 10000 – 20000 modes. Similar results were produced for the two other scenarios, but since they contained much fewer modes, only the data from the peak scenario are presented here.

![Figure 5.7: Execution time factor depending on mode count](image)

Several points from the graph are presented in Tab. 5.1 for a more precise overview of the performance speedup. Synthesis speed is defined as $1 / \text{execution time factor}$. In general, the speedup is higher when there are a large number of modes playing.

<table>
<thead>
<tr>
<th>Mode count</th>
<th>1 thread</th>
<th>8 threads</th>
<th>1 thread</th>
<th>8 threads</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>30000</td>
<td>0.23</td>
<td>0.15</td>
<td>4.41</td>
<td>6.82</td>
<td>55%</td>
</tr>
<tr>
<td>70000</td>
<td>0.76</td>
<td>0.42</td>
<td>1.32</td>
<td>2.38</td>
<td>80%</td>
</tr>
<tr>
<td>100000</td>
<td>1.06</td>
<td>0.60</td>
<td>0.94</td>
<td>1.65</td>
<td>76%</td>
</tr>
<tr>
<td>140000</td>
<td>1.42</td>
<td>0.78</td>
<td>0.71</td>
<td>1.29</td>
<td>82%</td>
</tr>
</tbody>
</table>

Table 5.1: Synthesis performance comparison between 1 and 8 threads.
Although we performed experiments using three different scenarios, the first scenario turned out to be the most relevant for the final results since the two other scenes had less modes and barely exceeded the real-time limit. The execution of these other scenarios were however still of importance; we can now conclude that the synthesis did not get notably slower over time even though the number of modes was very high. It cannot be known for sure whether any problems would arise over a much longer time span, but at least there is no imminent evidence for a slow down. Similarly, the big objects scenario did not indicate any slowdown caused when connecting a large amount of sound sources to the same object. Again, it does not guarantee the performance in a more extreme case, but in total these scenarios both support the feasibility of our approach.

From the results in chapter 5.4 Synthesis Performance many conclusions can be drawn. Firstly, the overall improvement gained from the multi-threading was up to 82%, when playing 130000 – 140000 modes with 8 threads, compared to using only one thread. The multi-threading gave less improvement when the number of modes was lower; 55% when playing 20000 – 30000 modes, but at 50000 modes and higher the speedup was always around 80%, and the execution time showed a clear linear dependency on the number of modes. Note that our synthesis implementation stores a list of all playing modes totally detached from the objects and sound source structure, which may have contributed towards this linearity.

The next observation was how the performance improved compared to the number of threads. It was seen that using only two threads still gave a great improvement over the single-threaded case, with an overall speedup around 30%. Increasing to four threads similarly increased the performance by 60%. Finally, eight threads gave a speedup around 80%. This was a very small improvement over four threads, which is likely since the processor used has only four hardware cores. The use of hyper-threading explains why there is any improvement at all. During infor-
nal studies we found indications that using 12 or more threads would decrease the performance slightly, but in the final measurements there was no significant difference between 8, 12 and 16 threads. There is no reason to believe that even higher thread counts would increase the performance since there are only eight logical cores, and load balancing is already handled well by the implementation. Rather, the threading overhead would lead to lower performance at some point.

It is also important to point out that even the single-threaded approach was capable of synthesizing up to 90000 modes in real-time, which may be enough in many cases. It was pointed out by Raghuvashti and Lin (2006) that a few hundred modes is enough to represent most materials using their proposed mode compression. Assuming an average of 200 sounding modes per sound source, this implies that we could synthesize 450 sounds simultaneously.

Bonneel et al. (2008) claimed their FD synthesis to be 4–8 times faster than the TD approach by Raghuvashti and Lin (2006). Since we did not implement the TD synthesis, and executed our experiments on more recent hardware, it is difficult to correctly compare our results with their approaches. However, assuming that our implementation of the FD synthesis proposed by Bonneel et al. (2008) was correctly done, which seems reasonable since both the performance and the quality of the generated sound were as expected, the results we present indicate that our multi-threaded approach significantly increases how many modes can be synthesized in real-time, over previous attempts.

A study by Cahill (2009) presents a multi-threaded approach to TD synthesis. Their experiments were also performed on an Intel Core i7. Although it is not the exact same processor as we used, the similar hardware architecture means that their results should be comparable to our study. Their multi-threading supposedly increased the performance by a factor 20, but in total their application could only successfully synthesize up to 10000 modes. In our study we synthesized up to 14 times more modes using only slightly faster hardware.
Chapter 7

Conclusions and Future Work

We have implemented sound synthesis in the frequency domain and contributed by adapting this into a multi-threaded approach. Measurements were done in three scenarios to find the relation between the performance, the number of modes and the thread count. Overall, we managed to answer both research questions with satisfactory results:

1. How much can multi-threading improve the performance of the frequency domain sound synthesis? Performance refers to the number of modes that can be synthesized in real-time.

It was found that using 8 threads improved the performance around 80% over the single threaded approach when a large number of modes were playing, on the Intel Core i7 processor used in the experiment. Using a single thread, 90000 modes were synthesized in real-time. With eight threads, 140000 mode shapes were synthesized 30% faster than real-time. This theoretically implies that up to 180000 modes would have executed in real-time if a scenario with even more sounds had been designed.

2. Is CPU-based multi-threaded sound synthesis viable for synthesizing contact sounds in a VE application?

Seeing that a very high number of modes could be synthesized in real-time it is clearly viable to use CPU sound synthesis in VE applications. Further, we found that even a single-threaded synthesis approach in the frequency domain can be viable to synthesize sounds.

Our results suggest that even a single-threaded synthesis can achieve sufficient performance for a VE application. Hence, it would be of great interest to perform further studies on how the synthesis can be guaranteed to stay within a fixed execution time. This topic has been partly covered by Raghuvanshi and Lin.
(2006) in terms of quality scaling, but could be further extended to incorporate multi-threading with the number of threads being varied dynamically as needed. It could for example be possible to temporarily allow much more modes, with more threads, to emphasize the sound realism during a cinematic cut-scene in a game.

With the ability to synthesize such a large number of modes, the next step would be to utilize this in order to improve the sound quality and realism. Many other techniques are available which could possibly be incorporated in the synthesis; friction-induced sounds by Avanzini et al. (2005), contact sounds between textured models by Ren et al. (2010) and example-guided material sounds by Ren et al. (2013), just to mention some.

With newer CPU generations trending towards having more cores, it is increasingly important to utilize more threads. The proposed multi-threaded implementation of FD synthesis was found to increase the performance greatly on more recent computer hardware. We think that sound synthesis remains a promising technique, and if only it can become more accessible through for example a third-party code library, it shows great potential for a more wide-spread use in commercial applications.
References


Appendix A

Glossary

Virtual environment (VE)
A computer-simulated virtual world where one or more users can interact in real-time, such as a video game or an educational simulator.

Rendering pipeline
The sequence of steps taken by a VE application to rasterize the virtual world into a 2D image representation. Sometimes also referred to as the graphics pipeline.

Mode
A mode of vibration, characterized by its frequency and phase offset, corresponding to a characteristic mode of vibration of the related object. In the context of contact sounds it also involves a decay rate.

Synthesis
Unless otherwise stated this refers to additive modal sound synthesis, which is the process of generating a sound by adding a number of modes.

Real-time Synthesis
The process of synthesizing a continuous audio stream within a real-time application, with mode parameters possibly changing over time.

Impact
A collision between two or more physically simulated objects in a VE application.
Contact Sound
A sound effect played as auditory feedback to signal when an impact has occurred.

Sounding object
A physically simulated object designed to trigger a contact sound upon impact.
Appendix B

FMOD Synthesis Setup

```cpp
#include "include/fmod.hpp"
#include <cstring>
#include <cmath>
#include <iostream>

FMOD::System* audioSystem;
FMOD::Sound* synthSound;
FMOD::Channel* synthChannel;
float time = 0.0f;

FMOD_RESULT FMOD_CALLBACK readDataCallback(
    FMOD_SOUND* sound, void* data, unsigned int datalen) {

    short* buffer = (short*) data;
    unsigned int numSamples = datalen / sizeof(short);

    for(unsigned int i=0; i < numSamples; i++) {
        buffer[i] = sinf(440 * 2 * 3.1416f * time) * 20000;
        time+= 1.0f / 44100.0f;
    }

    // Read from your buffer here...
    return FMOD_OK;
}

FMOD_RESULT FMOD_CALLBACK setPosCallback(
    FMOD_SOUND* sound, int subsound,
    unsigned int position, FMOD_TIMEUNIT postype) {

    // this callback is not used by the synthesis
    return FMOD_OK;
}
```
void initSynthesis() {

    // Init FMOD
    FMOD::System_Create(&audioSystem);
    audioSystem->init(16, FMOD_INIT_NORMAL, 0);

    // Init the synthesis channel
    FMOD_CREATESOUNDEXINFO audioParams;
    memset(&audioParams, 0, sizeof(FMOD_CREATESOUNDEXINFO));
    audioParams.cbuffersize = sizeof(FMOD_CREATESOUNDEXINFO);
    audioParams.decodebuffersize = 44100;
    audioParams.length = 44100 * sizeof(short) * 5;
    audioParams.numchannels = 1;
    audioParams.defaultfrequency = 44100;
    audioParams.format = FMOD_SOUND_FORMAT_PCM16;
    audioParams.pcmreadcallback = readDataCallback;
    audioParams.pcmsetposcallback = setPosCallback;

    FMOD_RESULT result = audioSystem->createStream(
        NULL, FMOD_OPENUSER | FMOD_LOOP_NORMAL,
        &audioParams, &synthSound);

    // Start the synthesis
    if (result == FMOD_OK) {
        audioSystem->playSound(synthSound, NULL, false, &synthChannel);
    }
}

int main() {
    initSynthesis();

    std::cout << "Press enter to quit ..." << std::endl;
    std::cin.get();
}