UNIVERSITY OF CALIFORNIA, SAN DIEGO

Ruratae: A physics-based audio engine

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Music

by

Andrew S. Allen

Committee in charge:

Professor Miller Puckette, Chair
Professor Thomas Erbe
Professor Edwin Harkins
Professor David M. Smith
Professor Tamara Smyth
Professor Rand Steiger

2014

The dissertation of Andrew S. Allen is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____

_____

_____
Chair

University of California, San Diego

2014

DEDICATION

During the process of writing this these, the encouragement and feedback I received from my colleagues, friends and family was immensely invaluable. I cannot thank those people enough for the patience and attentiveness during this time.

My thesis is dedicated to two very special people:

To Susanna R. Var,
who has supported me throughout writing this thesis,
and has given me a better understanding of myself.

And to my mother, Paula Davida Allen,
who has always been there for me,
and has selflessly supported me throughout my life.

To these two lovely woman, I owe an huge debt,
And without their moral and spiritual support, I would not be in the position I am today.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF SUPPLEMENTAL FILES

Video 1: Harp Example 1 (harp.1.mpg)

Video 2: Harp Example 2 (harp.2.mpg)

Video 3: Harp Example 3 (harp.3.mpg)

Video 4: Harp Example 4 (harp.4.mpg)

Video 5: Harp Example 5 (harp.5.mpg)

Video 6: Harp Example 6 (harp.6.mpg)

Video 7: Harp Example 7 (harp.7.mpg)

Video 8: Harp Example 8 (harp.8.mpg)

Video 9: Harp Example 9 (harp.9.mpg)

Video 10: Drum Example 1 (drum.1.mpg)

Video 11: Drum Example 2 (drum.2.mpg)

Video 12: Drum Example 3 (drum.3.mpg)

Video 13: Drum Example 4 (drum.4.mpg)

Video 14: Drum Example 5 (drum.5.mpg)

Video 15: Drum Example 6 (drum.6.mpg)

Video 16: Drum Example 7 (drum.7.mpg)

Video 17: Drum Example 8 (drum.8.mpg)

Video 18: Drum Example 9 (drum.9.mpg)

Video 19: Drum Example 10 (drum.10.mpg)

Video 20: Thing Example 1 (thing.1.mpg)

Video 21: Thing Example 2 (thing.2.mpg)

Video 22: Thing Example 3 (thing.3.mpg)

Video 23: Thing Example 4 (thing.4.mpg)

Video 24: Thing Example 5 (thing.5.mpg)

Video 25: Thing Example 6 (thing.6.mpg)

Video 26: Thing Example 7 (thing.7.mpg)

Recordings on file at Mandeville Special Collections Library

ACKNOWLEDGEMENTS

I would like to acknowledge Professor Tom Erbe for his support and practical advice as I devised the preliminary software experiments that led to this thesis. His advise and expertise in programming enabled the efficient software realization discussed in this thesis.

I would also like to acknowledge Professor Tamara Smyth for her tutelage in computational acoustics and her passion for learning and investigation. Her tenacity as a researcher and scholar has been a huge inspiration for my own work.

I would like to acknowledge the support of Professers Rand Steiger and Edwin Harkins, who have helped me grow through my time at the department and have given me the liberties neccessary to expand my knowledge and understanding.

I would also like to acknowledge Professor Davey Smith, who has always graciously offered me and my friends good advice and support. I know many people that are thankful for him and his generosity, kindness and understanding.

Finally, I would like to acknowledge Professor Miller Puckette, my committee chair, mentor and friend. Professor Puckette has always encouraged my various pursuits, compositional ideas, experiments, and research goals, always offering me his assitence when I'm in need. This thesis would simply not be what it is without his support.

VITA

2007	B. M. in Music Composition *magna cum laude*, Univeristy of South Carolina
2009	M. A. in Music Composition, The Eastman School of Music
2014	Ph. D. in Music, University of California, San Diego

ABSTRACT OF THE DISSERTATION

Ruratae: A physics-based audio engine

by

Andrew S. Allen

Doctor of Philosophy in Music

University of California, San Diego, 2014

Professor Miller Puckette, Chair

In this thesis, I will demonstrate the capabilities of Ruratae, a physics-based audio rendering engine that models and sonifies mechanical vibrations of Newtonian bodies. This new system will allow its users a range of possibilities and subtle controls without requiring expert knowledge of signal theory and acoustics. It will be an environment that allows users to produce dynamic, re-configurable, and interactive sounds through physically-intuitive construction and playing behaviors. I will discuss at detail known problems and issues that arise when attempting to afford these abilities, and I will offer several solutions and strategies that can be employed to tackle and reduce these upsets.

Chapter 1

Introduction

Through this document, I propose in detail a new, real-time physical audio simulation environment that is both powerful and accessible to the end-user. It will be an environment that allows users to produce dynamic, re-configurable, and interactive sounds through physically-intuitive construction and playing behaviors. The following sections outline desiderata that describes the intentions of this work. Fundamentally, three core demands must be met: that the model's representation and sonification is plausible, that the model is highly interactive and configurable, and that the model is robust, stable and re-configurable.

1.1 Plausibility

First and foremost, I want to develop a system that allows its users a range of possibilities and subtle controls without requiring expert knowledge of signal theory and acoustics. Instead, I will present a system that expands user understanding of the underlying principles out of their curiosity and subsequent experimentations and interactions with the system. The user should be able to produce easily repeatable results through their interactions with the system in order to cement an understanding of the system's behaviors. For this system to be successful, I require an interface that is highly plausible, with a strong correlation between the control parameters and the physical representation of the model.

To me, it seems self-evident that such an interface requires an appropriate

visual rendering of the structural objects and that this rendering should respond directly to the physical displacements depicted in the model. In addition to this vibrating depiction, the visual rendering must inform the user about the system's underlying properties. This information should be built directly into the visual representation, without resorting to table lookups or other tedious data representations. The visual representation can be altered and artistically re-interpreted, depending on context. If the system is being used in an end-user context, such as a live performance, art installation or digital media distribution, much of the underlying properties of the system can be hidden from view. If the system is being used within an editor or in a performance with no live video projection, than the system can provide a larger quantity of information to inform the user as they develop their model. In general, I wish to strive for as clean and minimal a representation as possible, without compromising the integrity of the correlation.

The methods of controlling the system should also be part of this correlation. They should be depicted in the visual representation and should assist the user in modifying the model's structural and physical properties. I also wish to optimize the number of controls; allowing users to develop both a sense of expression and mastery over their controls. Users should be able to create identical events that produce repeatable results, allowing them to observe gradual patterns of interaction over time exposed to the system. Users should not be overwhelmed by too many controls, I wish to reduce the number of ways users interact with the system but boast subtlety of control. It is important to find ways to consolidate control events into unified methods of operation, to emphasize depth of control variability over variety of controls.

1.2 Interactivity

Ideally, the system should run in real-time, in order to produce truly dynamic, responsive interaction. Although there is no reason why it could not benefit from being run on supercomputer clusters or with field-programmable gate arrays (FPGAs), optimizing the system for consumer hardware ensures portability and

extensibility. Enabling users with real-time functionality on cheaper components allows a larger distribution for a variety of creative projects. By reducing the computational costs of the system's algorithms and making its memory footprint relatively small, the system becomes more and more portable, enabling use in laptop performance, integration into pre-existing multimedia projects, use in custom hardware or Arduino chips, and so forth. For these reasons, the system should be able to function in real-time or near-real-time on conventional, modern consumer hardware. Because of this demand, a considerable effort has been placed on optimizing the system's routines while still preserving the plausible and robust expectations of the system.

Additionally, users should be able to use a multitude of input devices to interact. The system should be agnostic to input devices, allowing any current, past or future devices the potential to be interfaced with the system. Devices such as keyboards, triggers, infrared sensors, accelerometers, joysticks and microphones are just a few examples of the types of manners users may use to interact with the system. Of course, these are simply some of the types of interfaces that one might use (at least at the present time), but ideally this system's life can be prolonged to handle input from future technologies just as easily. There needs to be a unified adapter interface for these interactions that allows input to be generalized into parameters the system understands. Users should be able to effortlessly interact with the system with any and all of possible inputs in a completely configurable manner. Network capabilities should allow client users to interact with the system on a server-side as well. Ideally, it should be as immediately responsive as possible, limited only by the latencies of the input and output devices themselves.

A large emphasis is placed on user editing actions, in order to provide the user with the power to control and design their model to whatever specifications or inquiries they might have. Users should be able to easily modify any parameter of the system and the system should treat this in the same way across all parameters. Models should be able to be saved, restored, reset/undone, and instanced. Ideally, a user should be able to create a portion of the model and instance that portion several times to create a larger model. For example, a user might create a "string"

model that they then instance several times to create a multi-string instrument model.

1.3 Robustness

Stability and robustness are two vital components to any system intended for widespread use. Though users can expect and must tolerate small errors, any large instabilities will lead to undesirable results. It is important to take precautions to avoid amassing too much global error, which results in artificial damping, frequency drift, and worst of all, exponential growth.

Though it is discussed in much more detail later, it is impossible to overemphasize the importance of stability and energy preservation in computational physics models. As a precaution, the system should assist the user when creating and modifying models, keeping the properties they attempt to adjust within stable boundaries. In the case of an instability, the system should be able to recognize it as such and be able to auto-recover itself, either by gradually reducing energy back to stable levels or resetting to previous stable states. Within stable ranges, the system should try to model its interactions as accurately as possible, so that the user can reasonably "tune" the instrument and predict the spectral and durational content of their model.

Additionally, the system should be able to be reconfigured in real-time, allowing users to create, destroy and modify the model. Enabling this behavior has many interesting results, including designing models that account for hearing the sounds of destruction and deformation physics. Treating creation, destruction and modification physics in the same manner as performance (excitation) physics allows the system a high degree of interactivity and robustness. Allowing the model to be deformed, reshaped, torn, ripped and exploded allows the user a level of interaction distinct to this type of virtual instrument performance.

To summarize, the design desiderata of this document mandates a model that has a strong visual-aural-physical correlation, that is highly optimized for

real-time use, supports a range of input devices, and is robust to a variety of user interactions, preserving accuracy and energy. In the following chapters, I discuss the system's past, present and future state. In chapter one, I discuss the previous contributions of computer music that influenced this work, followed by chapter two's explanation of the model's mathematical frameworks. I then discuss the design specifications and implementation details in chapters four and five. Lastly, I discuss a real-world example of the system and the ramifications of such a system for real-time interactive media and video games in chapter six.

Chapter 2

Previous Contributions

Physical modeling is a well-researched and thoroughly-investigated discipline. Over several decades, many applications have arisen for modeling acoustical phenomena using a combination of signal processing and computational physics techniques. Some of these developments, such as digital waveguides [16], have enabled physically-responsive musical instruments to function on consumer hardware since the early 1990s. Finite element methods [5] have been an invaluable resource to engineers and acousticians in recent years, pushing the limits of computational power in a effort to asymptotically describe the incredibly intricate vibrations of large architectural structures and musical instruments alike. To suggest an alternative to an already incredible rich set of tools and methods would require an application suited to tasks not already absorbed by these immense disciplines. The work in developing CORDIS-ANIMA [6], which models physical vibration in mass-spring models, was largely absorbed into the development of computational physics engines of the past decade [14, pg. 2-7]. Though these engines can manage large sets of rigid and soft body dynamics (and to some sense, vibrations), they largely absolve themselves of the intricacies of acoustics, especially propagation. Furthermore, these engines are typically designed to analyze sub-audible structural soundness, measuring vibrations much lower than audible range [14, pg. 55]. The research presented here attempts to synthesize the work of computational physics engines and the interfaces computational acousticians are familiar with, to enable the construction, modification and destruction of audibly-vibrating bodies, in real-

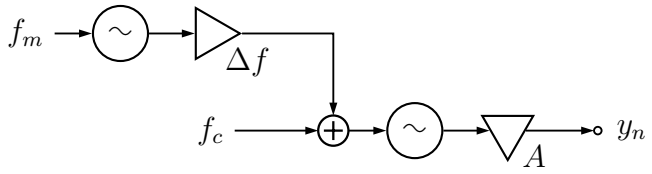

Figure 2.1: Example of Abstract (FM) Synthesis

time. This first chapter delves into the research goals, strengths and weaknesses of various physical-modeling techniques and how they compare to the alternative method presented in this dissertation.

2.1 Abstract Synthesis

Abstract synthesis [4, pg. 2-8] techniques such as additive, subtractive, wavetable, AM, and FM synthesis do not derive explicitly from physical behaviors but can be manipulated to produce convincing mimics. This class of techniques is canonical in the literature and has been researched thoroughly for the purposes of electronic music; A great amount of that research being for the modeling of established musical instruments.

A majority of these techniques were developed for computers of the 1970s and 80s, hence when adapted for modern systems, those technique run extremely efficiently, in terms of both computational and memory costs. Techniques such as wavetable synthesis may have a more significant memory load, depending on the number of tables used. Most abstract methods have guaranteed stability boundary conditions. Those that are not, such as IIR filters and recursive delay-line networks, can be stable when filter stability rules are met.

Though we refer to them as "abstract", in practice it is the case that every synthesis technique has behaviors appropriate to modeling different phenomena. In specific cases, these behaviors can be applied to physical processes. For example, FM synthesis (depicted in Fig. 2.1) is a non-linear method that makes it a desirable candidate for modeling the complex spectrum of many percussion and brass instruments. Additive or subtractive synthesis techniques assemble many

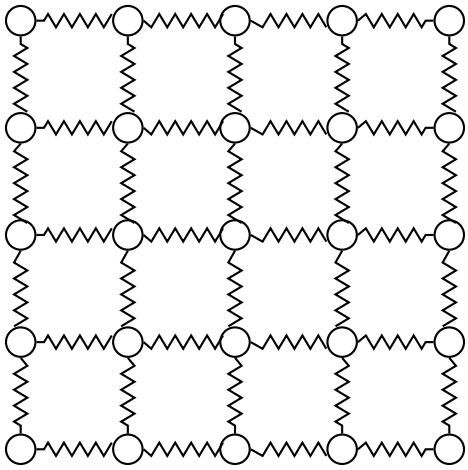

Figure 2.2: Network of masses connected by springs

small components to create aggregate timbres; the accuracy of these techniques increases with the number of components used. There are many cases where a priori knowledge regarding our model can allow us to construct very realistic models from abstract synthesis techniques. Unfortunately, abstract synthesis techniques do not intuitively map to physical models. Because they essentially seek accuracy in sound and not behavior, a model must undergo constant intervention.

Because it lacks any notion of physical representation, abstract synthesis also lacks any reasonable robustness. Aside from specific, case-by-case associations one might adopt between abstract synthesis parameters and physical phenomena, there are very few principles that can be directly implemented from model to model. These issues therefore make reconfiguration very tricky to pull off.

Since there is no physical accuracy and hence no robustness, it follows there cannot be any homogeneity. There is no way to determine a reasonable physical state and hence, any visual representation of abstract synthesis would be entirely the work of fiction; which isn't necessarily a bad thing, just less desirable for our purposes.

2.2 CORDIS-ANIMA

CORDIS-ANIMA was a software project developed by researchers at ACROE in France [6]. It is essentially a mass-spring system (an example of which is depicted in Fig. 2.2). This system, at its core, integrates systems of ordinary differential equations, solving an ODE for each mass particle in the system [4, pg. 8-10]. The power of working at this atomic level incurs a harsh penalty. ODEs that attempt to describe models with millions upon millions of particles are easily the most computationally expensive of all models listed here. One can reduce computational costs simply by approximating a structure using less particles, at the detriment of fluidity and accuracy of the model.

ODEs, like PDEs, depend on the proper numerical method and time step to remain stable. Even so, it is difficult to ensure global stability in an ODE system. Since ODEs individually integrate component particles, they can only account of local stability at any given particle. Because it is difficult to manage global, system-wide energy growth, it is very easy for that energy growth to go undetected in the system, allowing numerical errors to result in instability. Additionally, a proper step size can be difficult to determine in large ODE systems that are comprised of varying degrees of stiffness. Of course, for very complex behaviors like those found in fluid dynamics, this means that one would need to construct models of millions upon millions of particles to properly represent a given type of complex physical phenomena.

Because of their localized nature (integration occurs on one particle at a time), ODE systems are incredibly robust, even in real-time, and can easily account for destructive and constructive forces. In addition, because one can directly integrate forces on any given particle, it is possible to simulate nearly any imaginable excitation method such as plucking, striking, blowing or bowing; simulating them as Newtonian forces. Because ODE systems are built out of very elementary interactions, they allow for systems of almost any imaginable orientation to be simulated. For this reason, ODE systems are the popular choice of real-time interactive physics simulations found in games and interactive media.

ODE systems, like PDEs, operate directly on physical state information.

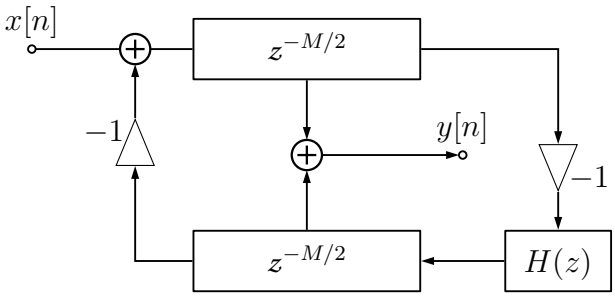

Figure 2.3: Example of Digital Waveguide Synthesis

Visual representation is, therefore, trivial to acquire.

I have examined several techniques, determining that, for physically-intuitive construction and behaviors, ODEs are the most suitable technique. Given their robustness, homogeneity and ability to generate very complex behaviors out of very simple rules, ODEs are significantly more well-equipped to handle the unknowable requests of real-time users. Though cost and stability are significant problems, several potential solutions arise and will be discussed in detail in the following chapters.

2.3 Digital Waveguides

Digital waveguides (depicted in Fig. 2.3) are a technique of abstract synthesis with applied physicality [4, pg.11-14]. The waveguide technique can be viewed as a form of subtractive synthesis with a feedback-delay. Because the technique is fundamentally structured in the same way as abstract synthesis techniques, it has many of the advantages of abstract synthesis with the addition of having some semblance of physical state. Waveguides model wave propagation through a medium, and it has been shown that the one-dimensional waveguides are indeed mathematically equivalent to the one-dimensional wave equation. The wave equation is a type of PDE that describes propagation waves through a medium of constant density and tension. This equation fundamentally describes many forms of physical vibrations and it is because of this that we use it often for simulating structures such as idealized acoustic tubes, strings, and drums.

Waveguides can be extended to higher dimensions by the use of a scattering junction [17], a port which models reflections and transmissions at impedance changes. Waveguides can also model inharmonic spectra by running multiple waveguides in synchronized, parallel bands, a technique known as banded waveguide synthesis.

One-dimensional waveguides are an extremely efficient means of simulating propagation through a constant medium. Because they are built from abstract synthesis techniques, they can be extended with additional abstract techniques while retaining their relatively low cost. As noted above, scattering junctions can be used to increase dimensionality, but at a sharply-increasing computational cost. While the 1-D waveguide is significantly cheaper than a 1-D PDE, the 2-D waveguide is equally expensive as the 2-D PDE, and a 3D waveguide is significantly more expensive that the equivalent 3D PDE.

Waveguides require the conventional filter stability conditions to remain stable. Like with abstract techniques, additional stabilizing processes can always be added for additional local stability.

Waveguides are ideal candidates for representing 1-D wave equations. Banded waveguides are competent at modeling inharmonic spectra and scattering junctions introduce additional normal modes and higher dimensionality. All of these techniques work well to model many idealized physical behaviors. However, if we desire the modeling of mediums with either irregular shape or non-constant density or tension, waveguides become less practical. Non-linearities can often still be modeled with the help of scattering junctions, though waveguide models with high amounts of scattering junctions tend to lose their cost advantages. A significant drawback to waveguides is their inability to accurately measure large displacements or destructive forces.

Scattering junctions make waveguide robustness possible. With care, waveguides can be reconfigured in real-time as long as energy in the scattering junction is properly handled. With enough scattering junctions, waveguides can function much like any PDE or ODE system of wave propagation, though this requires significant programmer intervention to implement adequate dynamic junctioning.

One can derive an approximate physical state of any point along a waveguide by trivial signal operations. However, it should be noted that this derivation is not entirely accurate as it does not evaluate large displacements; systems that occasionally deal with very dynamic forces will therefore not be accurately represented. Additionally, a waveguide exists over some length of space-time, which at every point along this line exists an equilibrium state and some displacement above and below this equilibrium. It follows that a waveguide occupies an undefined 2-D plane in dimensional space. The transformation of this plane, (its scale, rotation and translation), are not explicitly part of the waveguide model. Thus, the plane's transform is undefined, meaning that 1-D waveguides cannot be mapped in N-dimensional space without ambiguity. To complicate things further, scattering junctions between these segments have no inherit sense of direction, and therefore they have no explicit dimensionality either. It is very important that the programmer of such a system take special care to make the relationship between visual and audible physical state consistent and unambiguous.

2.4 Modalys

Modalys is a piece of software developed by researchers at IRCAM [11]. It uses modal decomposition and synthesis to determine linear and non-linear modes of vibrations for rigid bodies. Modal synthesis is another technique that approximates the solution to simple PDEs. Modal synthesis has many similar advantages to waveguide synthesis but instead of approximating a solution to the physical state of a PDE, modal synthesis approximates the solution to the PDE in terms of the energy present as each normal mode (a normal mode is the physical location of a resonant frequency of the model). The first step in modal synthesis is to perform an eigendecomposition on the PDE to represent the system in terms of its normal modes. Once a matrix of normal modes has been established, initial conditions can be provided and the system is numerically integrated over time; at each time step, the frequency and amplitude for each normal mode is produced. This information is usually then supplied to a bank of oscillators to generate the

timbre of the structure at the given point in time.

The cost involved in using modal synthesis comes from the eigendecomposition of the PDE (assuming one is possible) and from numerical integration for each mode at each point in time [4, pg.10-11]. Therefore, the complexity of the problem's PDE as well as the number of modes one wishes to track will determine the relative cost of the technique. For simple models based on wave equations, the efficiency is on par with waveguides. However, some PDEs cannot be diagonalized, and therefore it may be impossible to deduce the eigendecomposition for some meshes. In order to determine the normal modes of these sorts of systems, very expensive integration methods and spectral analysis must be computed prior to beginning integration of the technique's output.

When eigendecomposition is possible, modal synthesis is typically stable. However, if an eigendecomposition is not possible and direct simulation of the PDE is the only way to determine the normal modes, numerical integration errors can arise depending on method. With enough error, the system is prone to instability.

Modal synthesis accuracy is determined by the number of modes modeled and the accuracy of the initial PDE system in describing a given model. Similar to waveguides, a limitation of modal synthesis is the assumption that the model in question does not undergo large displacements that would result in large changes to modal frequencies. Likewise, modal synthesis systems do not model high energies.

Any reconfiguration in the system will require another eigendecomposition. Real-time configuration should be possible, but arbitrary meshes can be unpredictably expensive (and sometimes impossible). Because modal synthesis cannot accurately model large displacements resulting from high energies, a model that disassembles itself is not feasible without significant programmer intervention. Since modal synthesis derives its solution solely from the frequency-domain, there is no direct visual representation of physical state, but it can be deduced through analytical means.

2.5 Finite Element Methods (FEMs)

Finite element methods (FEMs) [4, pg. 16-18] use triangle meshes to quantize the topology of a model in order to numerically solve for forces along the mesh. FEMs typically use partial differential equations to approximate forces acting on a model. PDEs enable one to directly simulate a PDE system and to avoid the pitfalls of approximation techniques like waveguides or modal synthesis. Of course, direct simulation of PDEs has its own pitfalls: high computational costs and the potential for numerical instability.

The entire reason that waveguides and modal synthesis exist is that direct PDE simulation is very computationally expensive, except in the few special cases when PDEs actually perform better than their counterparts (i.e. 2-D (or higher) PDEs perform better than 2-D (or higher) waveguides **??**, PDEs perform better than modal synthesis with complex mesh topographies). The high costs are not unmanageable and can often be optimized significantly for fundamental PDEs. The 2-D wave equation is an example of a highly optimized PDE that is used often in computer graphics to simulate surface waves in liquids.

Stability conditions for PDEs are dependent on the accelerations produced by the PDE's variables. By choosing the proper numerical method to integrate by, as well as choosing the right time step to integrate at, most PDEs remain stable. Differential equations that produce very large acceleration values are consider stiff equations and require very small time steps and very specialized numerical methods in order to retain their stability.

Almost any well-defined physical propagation (such as airflow, heat dispersion, or surface vibration) can be described as a system of PDEs and therefore, a PDE-based model accurately simulates almost any flow-based physical behavior.

PDEs can be reconfigured by re-computing any variables in the equation. In the real-time case, care must be taken to avoid instabilities from sudden changes in the PDE's variables or otherwise new variables may require a smaller time step.

Directly simulated PDEs operate on physical state. Choosing a preferred data representation, be it audible or visual, is trivial.

Chapter 3

A Mathematical Framework for Vibration

The physics engine proposed in this work models sonic vibrations as mechanical oscillations of atomic particles. The particles are connected together by attracting and repelling forces, and when organized into a network, produce rich and dynamic spectra. I will build this model up, starting with the simplest forms of oscillation and working into more and more intricate systems, beginning with simple harmonic motion.

3.1 Simple Harmonic Motion

Simple harmonic (sinusoidal) motion is the simplest form of periodic oscillation [8, pg. 4-6]. This motion is describe by

$$x = A \sin\left(2\pi \frac{t}{\tau} + \phi\right), \tag{3.1}$$

where x is displacement, τ is the period of the oscillation, t is time, ϕ is the phase offset, and A is the oscillation's amplitude.

When considering harmonic motion as being represented as moving along a circle at constant speed, I can designate angular frequency by ω, displacement x can be written as

$$x = A \sin(\omega t), \tag{3.2}$$

where

$$\omega = \frac{2\pi}{\tau} = 2\pi f. \tag{3.3}$$

The velocity and acceleration can be differentiated from (3.12), obtaining

$$\begin{aligned} \dot{x} &= \omega A \sin(\omega t + \pi/3), \\ &= \omega A \cos(\omega t), \end{aligned} \tag{3.4}$$

and

$$\begin{aligned} \ddot{x} &= \omega^2 A \sin(\omega t + \pi), \\ &= -\omega^2 A \sin(\omega t), \\ &= -\omega^2 x. \end{aligned} \tag{3.5}$$

This system is represented in the complex plane by using Euler's identity [8, pg. 6-7], $e^{i\theta} = \cos(\theta) + i\sin(\theta)$, substituting the values from the general solution

$$\begin{aligned} z &= Ae^{i\omega t}, \\ &= A\cos(\omega t) + iA\sin(\omega t), \end{aligned} \tag{3.6}$$

where z is a complex sinusoid.

3.2 Mechanical Illustration

The core of the engine is focused on Newtonian dynamics. Let's begin by considering the elementary case of a mass connected to a wall by a spring, in one dimension [8, pg. 4].

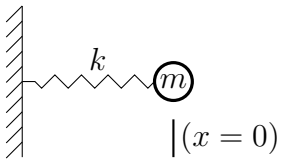

Figure 3.1: A mass at equilibrium connected by a spring to a fixed wall

Figure 3.1 shows a mass, m, attached to a wall via a spring with stiffness k; the mass is displaced by x. The spring acts to maintain an equilibrium between the mass and the wall.

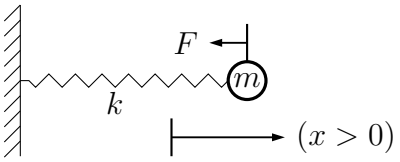

Figure 3.2: A mass, displaced from its equilibrium, connected by a spring to a fixed wall

Displacing the mass results in a restoring force from the spring, pull back in the opposite direction of its displacement (depicted in Fig. 3.2). This reciprocal force describes Hooke's law, which is formalized as

$$F = -kx. \tag{3.7}$$

3.3 Classical Definition

When applying Newton's second law, $F = ma$,

$$m\ddot{x} = -kx, \tag{3.8}$$

$$m\ddot{x} + kx = 0. \tag{3.9}$$

The angular frequency, ω_n, is defined by the equation

$$\omega_n = \sqrt{\frac{k}{m}}. \tag{3.10}$$

(3.9) can be written as

$$m\ddot{x} + \omega_n^2 x = 0, \tag{3.11}$$

a homogeneous second-order linear differential equation, which has the general solution

$$x = A\sin(\omega t + \phi), \tag{3.12}$$

or

$$x = \dot{x_0}\cos(\omega_n t) + x_0 \sin(\omega_n t), \tag{3.13}$$

where x_0 is initial displacement and $\dot{x}_0$ is initial velocity. The amplitude and phase are derived from the system's initial conditions

$$\begin{aligned} A &= \sqrt{\dot{x_0}^2 + x_0^2}, \\ \phi &= tan^{-1}\left(\frac{\dot{x}_0}{x_0}\right), \end{aligned} \tag{3.14}$$

3.4 Damped Harmonic Motion

Viscous damping force [8, pg. 11-13] is expressed by the equation

$$F = c\dot{x}, \tag{3.15}$$

where c is the damping constant. The damping ratio between damping constant and spring constant is defined [1, pg. 137-143] as

$$\zeta = \frac{c}{2\sqrt{mk}}. \tag{3.16}$$

Adding the damping force to (3.9), the equation of damped harmonic motion becomes

$$m\ddot{x} + c\dot{x} + kx = F(t). \tag{3.17}$$

When $F(t) = 0$, our equation reduces to a homogeneous differential equation which has a solution of the form

$$x = e^{\gamma t}, \tag{3.18}$$

and substituting terms ω_n, γ, and ζ into the characteristic equation

$$\gamma^2 + 2\zeta\omega_n\gamma + \omega_n^2 = 0, \tag{3.19}$$

which has two roots,

$$\gamma_{1,2} = -\zeta\omega_n \pm \sqrt{\zeta^2\omega_n^2 - \omega_n^2}. \tag{3.20}$$

This results in the general solution given by

$$x = Ae^{\gamma_1 t} + Be^{\gamma_2 t}, \tag{3.21}$$

where A and B are coefficients evaluated from the system's initial conditions:

$$A = x_0 + \frac{\gamma_1 x_0 - \dot{x}_0}{\gamma_2 - \gamma_1}, \tag{3.22}$$

$$B = \frac{\gamma_1 x_0 - \dot{x}_0}{\gamma_2 - \gamma_1}. \tag{3.23}$$

Depending on the value of ζ (and hence, the roots to the characteristic equation), the system will have one of three possible states: critically damped, over-damped, or under-damped [18].

3.5 Critically-Damped ($\zeta = 1$)

Critically-damped systems converge to zero as quickly as possible without oscillating. The general solution for a critically-damped system is

$$x(t) = (A + Bt)\, e^{-\omega_n t}, \tag{3.24}$$

where

$$A = x_0, \tag{3.25}$$

$$B = \dot{x}_0 + \omega_n x_0. \tag{3.26}$$

3.6 Over-damped ($\zeta > 1$)

Over-damped systems converge to zero slower than critically-damped systems. The general solution for an over-damped system is

$$x(t) = Ae^{\gamma_1 t} + Be^{\gamma_2 t}, \tag{3.27}$$

where

$$A = x_0 + \frac{\gamma + x_0 - \dot{x}_0}{\gamma_2 - \gamma_1}, \tag{3.28}$$

$$B = \frac{\gamma + x_0 - \dot{x}_0}{\gamma_2 - \gamma_1}. \tag{3.29}$$

3.7 Under-damped $(0 \leq \zeta < 1)$

Under-damped systems converge to zero slower than critically-damped systems while still oscillating. The frequency at which these systems is known as the damping angular frequency (ω_d). The general solution for an under-damped system is

$$x(t) = e^{-\zeta\omega_n t}\left(A\cos\left(\omega_d t\right) + B\sin\left(\omega_d t\right)\right), \tag{3.30}$$

where

$$\omega_d = \omega_n\sqrt{1-\zeta^2}, \tag{3.31}$$

$$A = x_0, \tag{3.32}$$

$$B = \frac{\zeta\omega_n x_0 + \dot{x}_0}{\omega_d}. \tag{3.33}$$

When $\zeta = 0$, the system is undamped and the damping angular frequency is the natural angular frequency.

3.8 Coupling, Normal Modes and Coordinates

Because I wish to model multi-frequency spectrum, I will need to deal with systems capable of producing more than one natural frequency. When masses are coupled together [8, pg. 34-36], they produce respective forces on one another and produce additional modes of vibrations (and hence, additional natural frequencies). The number of modes of a system is equal to the number of degrees of freedom in a system. Each normal mode of vibration has a unique configuration (normal coordinates) that corresponds to that mode's pattern of oscillation through the system. As I say above for single mode systems, determining the angular frequency required solving the linear differential equation. The linear normal modes and coordinates for undamped systems are determined through similar means, by solving for a linear system of differential equations. This can be achieved easily through matrix decomposition methods.

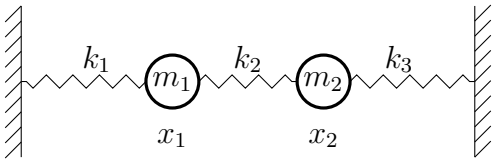

Figure 3.3: A two-mass at equilibrium connected by springs to two fixed walls

3.8.1 Eigendecomposition

As an example, suppose there are two masses connected together and to two walls via three springs in one-dimension [8, pg. 26-28] as shown in figure 3.3.

From the given parameters, a linear system of equations is formed that can solve for the system's normal modes and coordinates.

The equations of motions for the two masses are as follows:

$$\begin{cases} m\ddot{x_1} + k_1 x_1 + k_2(x_1 - x_2) = 0 \\ m\ddot{x_2} + k_2(x_2 - x_1) + k_3 x_2 = 0. \end{cases} \tag{3.34}$$

In order to solve for the modes of the system, the solution $x(t) = e^{i\omega t}$ is assumed, where $\ddot{x} = -\omega^2 x$. For this, the system should take on the matrix form

$$\frac{\sum_{n=0}^{N} F(x_n)}{m_a} = \ddot{x}_a, \tag{3.35}$$

$$\begin{bmatrix} -\dfrac{k_1 + k_2}{m_1} & \dfrac{k_2}{m_1} \\ \dfrac{k_2}{m_2} & -\dfrac{k_2 + k_3}{m_2} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} \ddot{x}_1 \\ \ddot{x}_2 \end{bmatrix}. \tag{3.36}$$

In order to simplify the solution for example purposes, let us assume the system is homogeneous, where $k = k_1 = k_2 = k_3$, $m = m_1 = m_2$ and

$$\alpha = -\frac{2k}{m}, \tag{3.37}$$

$$\beta = \frac{k}{m}. \tag{3.38}$$

Substituting for α and β:

$$\begin{bmatrix} \alpha & \beta \\ \beta & \alpha \end{bmatrix} \mathbf{x} = \ddot{\mathbf{x}}. \tag{3.39}$$

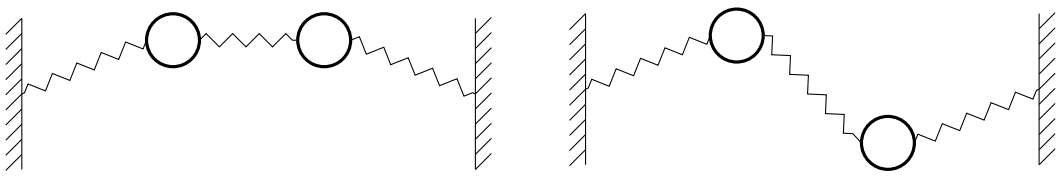

Figure 3.4: Normal coordinates $\mathbf{x}_{1,2}$ for $\omega_{1,2}^2$

Then, assuming a solution very similar to that of a single mass differential equation

$$\mathbf{x} = \mathbf{v}e^{i\omega t} = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} e^{i\omega t}, \tag{3.40}$$

and therefore

$$\ddot{\mathbf{x}} = -\omega^2 \mathbf{v}e^{i\omega t} = -\omega^2 \mathbf{x}, \tag{3.41}$$

$$\begin{bmatrix} \alpha & \beta \\ \beta & \alpha \end{bmatrix} \mathbf{x} = -\omega^2 \mathbf{x}. \tag{3.42}$$

$\mathbf{M}\mathbf{x} = \lambda \mathbf{x}$, where $\lambda = -\omega^2$,

$$\left|\mathbf{M} + \omega^2 \mathbf{I}\right| = 0 = \begin{vmatrix} \alpha + \omega^2 & \beta \\ \beta & \alpha + \omega^2 \end{vmatrix}. \tag{3.43}$$

The characteristic equation of the determinant is found to be

$$\omega^4 + 2\alpha\omega^2 + \alpha^2 - \beta^2 = 0, \tag{3.44}$$

which has the roots

$$\omega_{1,2}^2 = \frac{-2\alpha \pm \sqrt{(2\alpha)^2 - 4(\alpha^2 - \beta^2)}}{2} = -\alpha \pm \beta. \tag{3.45}$$

To find the eigenvectors (and hence the normal coordinates), the system is solved for $\mathbf{x}$, using the eigenvalues in the form

$$\left(\mathbf{M} + \omega_{1,2}^2 \mathbf{I}\right) \mathbf{x}_{1,2} = 0, \tag{3.46}$$

where $\mathbf{x}_{1,2}$ are the eigenvectors of the matrix.

Solving the eigenvectors using arbitrary constants,

$$\begin{vmatrix} \alpha + \omega_1^2 & \beta \\ \beta & \alpha + \omega_1^2 \end{vmatrix} \mathbf{x}_1 = \begin{vmatrix} -\beta & \beta \\ \beta & -\beta \end{vmatrix} \mathbf{x}_2, \tag{3.47}$$

$$x_{1,1} = x_{1,2}, \tag{3.48}$$

and

$$\begin{vmatrix} \alpha+\omega_2^2 & \beta \\ \beta & \alpha+\omega_2^2 \end{vmatrix} \mathbf{x}_2 = \begin{vmatrix} \beta & \beta \\ \beta & \beta \end{vmatrix} \mathbf{x}_1, \tag{3.49}$$

$$x_{2,1} = -x_{2,2}, \tag{3.50}$$

where, with normalized directions,

$$\mathbf{x}_1 = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}, \tag{3.51}$$

and

$$\mathbf{x}_2 = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix}. \tag{3.52}$$

resulting in the normal coordinates shown in figure 3.4. This formulation is particularly helpful in determining the normal modes of vibration for arbitrary linear systems of any number of masses.

3.9 Driven Vibrations

Normal modes describe the central frequencies at which the system tends to resonate [8, pg. 21-23]. A mode tends to have some amount of bandwidth around the central frequency. When driving a linear system, its important to take into account the spectrum of the driving force. When the driving force has a spectrum that is harmonically similar to the system, it tends to reinforce these central frequencies. However, when the driving force is harmonically dissimilar to the system, it tends to force the modes of the system to oscillate slightly above or below their natural frequency. Careful attention can be placed on deciding qualities of the driving force in order to produce dynamic spectral results from the system.

3.10 Non-Linear Vibrations

So far I have discussed purely linear systems, but it is the intention of this work to handle non-linear systems as well. Non-linear systems are simply systems

that have dependent variables that cannot be solved for [8, pg. 28-29]. For these reasons, eigendecomposition becomes an inefficient means of determining solutions and numerical integration from known initial conditions becomes the only possible avenue. Examples of non-linearity in the proposed system include collision forces (which are conditional forces based on agent proximity) [14, pg. 118-129], hysteresis (which is a non-linear state of springs) [13] and rigidness of springs (which produces a non-linear form of chaotic motion, similar to the motion of double pendulums) [2], discussed in the following chapter.

Chapter 4

Design Specification

4.1 Primitives

The audio-physics engine is effectively a mass-spring network, which consists of agents that interact with their neighbors using simple, relational behaviors. From these simple behaviors, the lowly agent-to-agent communications give rise to larger group dialogues; complex audible phenomena akin to the sounds of the physical world. I consider the lowest-level components of the system as its *primitive* types. These types are placed into one of two categories: the vibrating agents, and the forces that act on these agents. These primitives consist of *particles*, *springs*, *dampers*, *collisions*, *constraints* and *external forces*.

4.1.1 Particles

In this system, I refer to all vibrating agents as *particles*. A particle is the atomic unit of the system, consisting of a position vector ($\mathbf{x}_p$), velocity vector ($\dot{\mathbf{x}}$), mass (m_p), spherical volume (r_p) and kinetic friction (μ_p).

Position and Velocity

For strong plausibility of the model, I should be able to have a correlating visual model and be able to dynamically alter the configuration of the model. Though a Hamiltonian system ensures energy preservation, a Newtonian model

allows us to describe the system much more easily, and with re-configurable terms. Therefore, it is suitable to use a Newtonian state for each particle in the system.

Mass

If I ignored individual differences in mass in the system, the system would only exhibit homogeneous macro-behaviors which can reduce to a variation of the wave equation. By treating each particle's mass independently, the system can also exhibit non-homogeneous, dispersive behavior.

Spherical Volume

Because I wish to account for collisions in the system, I include a simple notion of volume. The engine considers particles to act as simple spheres, therefore the volume is treated as a radius. During collision detection, the distance between two particles is checked against sum of the two particles' radii to determine if any intersection occurs.

Kinetic Friction

The particle's kinetic friction value is the kinetic friction along any discrete point along the particle's circumference. The values from two particles are used to compute their restitution factor (relative energy loss) during an inelastic collision.

4.1.2 Springs

Springs produce harmonic motions through the particle network. Typically, springs are treated linearly. However, for the model, spring exhibit non-linear plasticity as well as linear elasticity. The restoring forces they produce both depend and act on pairs of particles, the displacements of which are denoted by $\mathbf{x}_a$ and $\mathbf{x}_b$.

$$\mathbf{F}_{spring} = -k_{spring}(\|\mathbf{d}\| - L_s)\frac{\mathbf{d}}{\|\mathbf{d}\|}, \tag{4.1}$$

where

$$\mathbf{d} = \mathbf{x}_a - \mathbf{x}_b, \tag{4.2}$$

$$k_{spring} = \begin{cases} \kappa_s, & \text{if } \|d\| < Y_s \\ \kappa'_s, & \text{if } Y_s < \|d\| < U_s \\ \kappa''_s, & \text{if } U_s < \|d\| < R_s \\ 0 & \text{if } \|d\| > R_s \end{cases} . \tag{4.3}$$

In addition to referencing the pair of the particles, a spring consists of an elastic constant (κ_s), a resting length, or equilibrium, (L_s), a yield point (Y_s), the spring's ultimate strength (U_s), the spring's rupture point (R_s), and two plastic constants (κ'_s and κ''_s).

Elastic Constant

The elastic constant describe the slope between stress and strain of the spring while it persists in its elastic region.

Resting Length

The spring produces a force relative to the displacement from its resting length between two particles. By introducing springs with non-zero resting length, systems have deformable shape. With sufficiently stiff springs, this shape becomes rigid, enabling the system to model vibrations in semi-rigid bodies. As the spring is displaced into the super-elastic region, the resting length is increased to account for the maximum stress placed on the spring. The engine can also allow the spring's resting length to decrease if the spring is compressed below its elastic region.

Yield Point

The yield point describes the point at which the spring becomes super-elastic, entering a region of plasticity, which often enacts a non-linear response in the spring dependent on how far displaced it is beyond this yield point. As the spring displaces further above the yield point, its resting length and yield point is increased, thus shifting its elastic region as well. As the spring releases from its maximum displacement above the yield point, instead of returning to its previous state, it now exhibits elastic behavior from this new yield point. At this point, the

spring may continue to increase the yield point until its point of rupture. At any time before this rupture, if the spring is displaced below the original yield point, its resting length and current yield point will decrease, which can enable it to return to its original state given the right conditions.

Ultimate Strength Point

The ultimate strength point is a point between the yield point and rupture point in the plastic region. Typically, the ultimate strength point is the point at which the spring produces the most stress for any given strain. The name is adopted out of convention, but depending on the spring's parameters, it may or may not exhibit this behavior, and more importantly acts as a point to break up the plastic region's slope into two linear segments.

Rupture Point

The rupture point is the point at which the spring will break, severing the bond between the two referential particles and removing itself from the system. Beyond the point of rupture, the spring cannot recover.

Plastic Constants

Plastic constants, like elastic constants, describe stress-strain slope, but in the plastic region of the spring. When modeling a ductile material, one might set the first plastic constant to have a lower slope than the elastic region and the second constant to have negative slope as it arrives at the rupture point.

4.1.3 Dampers

Like springs, *dampers* produce forces that both depend and act on a pair of particles. However, unlike springs, dampers are only treated in the linear case.

$$\mathbf{F}_{damping} = -r\left(\dot{\mathbf{x}}_a - \dot{\mathbf{x}}_b\right), \tag{4.4}$$

where

$$\mathbf{d} = \mathbf{x}_a - \mathbf{x}_b. \tag{4.5}$$

Aside from references to two particles, they consist simply of a damping constant (r). Though dampers are usually lumped into a damped-spring, I isolate damping in this system so that I can dampen particles that do not share spring forces. This is helpful when attempting to filter specific harmonics of a system (see chapter 4).

Damping Constant

The damping constant determines the amount of resistive force produced between the pair. Often dampers and springs act on the same particles. In those cases, it is useful to derive the damping constant from the damping ratio (ζ), which is relative to the spring's linear stiffness constant (k),

$$\zeta = \frac{r}{2\sqrt{km}}. \tag{4.6}$$

When $0 < \zeta < 1$, the spring will be under-damped. If $\zeta = 1$, the spring will be critically damped. If $\zeta > 1$, the spring will be over-damped.

4.1.4 Collisions

Collisions are conditional forces between pairs of colliding particles. A collision's force is determined entirely by its two particle references. It makes use of every component of both particles.

Elastic Collisions

Elastic collisions are collisions between two particles that conserve momentum (i.e. no energy loss due to friction) [14, pg. 118-129]. The conservation of momentum demands that the total momentum before the collision is the same as the total momentum after the collision

$$m_a\dot{\mathbf{x}}_a + m_b\dot{\mathbf{x}}_b = m_a\dot{\mathbf{x}}'_a + m_b\dot{\mathbf{x}}'_b, \tag{4.7}$$

where $\dot{\mathbf{x}}$ is velocity before the collision and $\dot{\mathbf{x}}'$ is velocity after the collision.

From this equation, I can derive the after-collision velocities for each particle from their before-collision velocities and masses.

$$\dot{\mathbf{x}}'_a = \frac{\dot{\mathbf{x}}_a\left(m_a - m_b\right) + 2m_b\dot{\mathbf{x}}_b}{m_a + m_b}, \tag{4.8}$$

$$\dot{\mathbf{x}}_b' = \frac{\dot{\mathbf{x}}_b (m_b - m_a) + 2m_a \dot{\mathbf{x}}_a}{m_a + m_b}, \tag{4.9}$$

only when $\|\mathbf{x}_a - \mathbf{x}_b\| < (r_a + r_b)$.

Elastic collisions suffice for systems of high-density, small mass particles, but can appear unrealistic for larger structures.

Inelastic Collisions

Inelastic collisions are collisions that do not conserve momentum by accounting for some energy loss due to friction between the two colliding particles [14, pg. 118-129]. The amount of loss is determined by the coefficient of restitution $C_r = \mu_a \mu_b$. The equations for velocity after inelastic collisions are defined as

$$\dot{\mathbf{x}}_a' = \frac{C_r m_b (\dot{\mathbf{x}}_b - \dot{\mathbf{x}}_a) + m_a \dot{\mathbf{x}}_a + m_b \dot{\mathbf{x}}_b}{m_a + m_b}, \tag{4.10}$$

$$\dot{\mathbf{x}}_b' = \frac{C_r m_a (\dot{\mathbf{x}}_a - \dot{\mathbf{x}}_b) + m_a \dot{\mathbf{x}}_a + m_b \dot{\mathbf{x}}_b}{m_a + m_b}, \tag{4.11}$$

only when $\|\mathbf{x}_a - \mathbf{x}_b\| < (r_a + r_b)$.

The advantage to this formulation is that the type of collision will vary depending on C_r. When $C_r = 1$, the additional friction calculation is omitted and the collision is perfectly elastic. When $C_r = 0$, the collision is perfectly inelastic, resulting in the particles "sticking" together, moving together at the same speed and direction,

$$\dot{\mathbf{x}}_a' = \dot{\mathbf{x}}_b' = \frac{m_a \dot{\mathbf{x}}_a + m_b \dot{\mathbf{x}}_b}{m_a + m_b}. \tag{4.12}$$

Whenever $0 < C_r < 1$, the collision is partially inelastic, which is what is commonly expected. Using inelastic collisions in the engine allows all types of collisions to occur during simulation. Since C_r is calculation from each particles' kinetic friction, the elasticity of the collision is entirely under the user's control. Typically, the engine will use Eqs. 4.10 and 4.11 for particle collisions unless the engine is explicity using only elastic collisions.

4.1.5 Constraints

Constraints [14, pg. 101-109] act between two particles, keeping them a given distance apart, to produce fully rigid bodies. One may think of a constraint

as a spring with infinite stiffness and a given resting length. It references a pair of particles and consists solely of a resting length (L_c),

$$\mathbf{x}_a = \mathbf{x}_a - \mathbf{o}, \tag{4.13}$$

$$\mathbf{x}_b = \mathbf{x}_b + \mathbf{o}, \tag{4.14}$$

where

$$\mathbf{o} = (\|\mathbf{d}\| - L_c)\frac{\mathbf{d}}{\|\mathbf{d}\|}, \tag{4.15}$$

$$\mathbf{d} = \mathbf{x}_a - \mathbf{x}_b. \tag{4.16}$$

4.1.6 World Forces

External forces are all the forces, model-driven and user-driven, that exist in the system but are not produced by the system's particles. Model-driven forces such as gravity ($\mathbf{g}$), air drag (b), and wind velocity ($\mathbf{W}$) can be treated like constants that exhibit predictable behavior over the entire system. Even when dynamically altered, these forces will generate fairly predictable behavior. However, user-driven forces such as microphone and keyboard input are inherently non-constant.

Gravity

The engine uses a simplistic model of gravity by applying a directional force to all particles in the system [12, pg. 390-394],

$$\mathbf{F}_{gravity} = m\mathbf{g}. \tag{4.17}$$

Air Drag

I can model the viscosity of the medium the model exists in (be it air, water or jello) by considering a force relative to the given particle's velocity [12, pg. 394-396],

$$\mathbf{F}_{drag} = -b\dot{\mathbf{x}}. \tag{4.18}$$

Wind Force

Wind can be modeling by applying a force relative to the direction of the difference between a particle's velocity and wind velocity [12, pg. 460]

$$\mathbf{F}_{wind} = k_{wind}|\left(\dot{\mathbf{w}} - \dot{\mathbf{x}}\right) \cdot \mathbf{N}|, \tag{4.19}$$

where $\dot{\mathbf{w}}$ is wind velocity and N is the normal of the particle. The normal can be calculated by determining the "edges" of the model's mesh from neighboring particles.

4.1.7 External Forces

Users can also submit their own force functions, these forces are accounted in total net force for the system.

4.1.8 Microphones

The engine would not be complete without a method for extracting sound. I treat the speed of sound as infinite, and simply measure directional velocity towards a listening position. The engine attenuates the signal based on the particle's distance from the listening position. This effectively models an omni-directional microphone. Additionally, by attenuating this velocity energy along the direction angle, the engine can model more complex pickup patterns such as bi-directional and cardioid patterns,

$$y = \sum_{i=0}^{N} \frac{\dot{\mathbf{x}}_i \cdot \left(\frac{\mathbf{d}}{\|\mathbf{d}\|}\right)}{\|\mathbf{d}\|}, \tag{4.20}$$

where y is the pressure produced by all particles in the system and

$$\mathbf{d} = \mathbf{x}_{mic} - \mathbf{x}_i. \tag{4.21}$$

A more accurate but complex listening model would also include a delay based on distance,

$$z = \frac{\|\mathbf{d}\| f_s}{c}, \tag{4.22}$$

where z is the delay in samples, f_s is the sample-rate, c is the speed of sound in the medium. Discretization errors will occur when particles move around, so each particle's delay should use 4-point interpolation as well.

4.2 Finite Difference Methods

In order to observe the state of the ODE system, and hence, "listen" to it, I must solve the system using finite difference methods. A finite difference method approximates a discrete solution to Newton's equations of motion with continuous forces [10, pg. 637]. These methods discretize the time domain of the differential equation. Once the domain is discretized, the methods produce a set of discrete numerical approximations to the derivative.

There are many finite difference methods that each have unique strengths and weaknesses. This article will cover a few basic types and from comparison, make a strong case for Beeman's Algorithm as the primary method for the engine. To simplify discussion, I will present each method as a solution to the motion of a particle in one dimension. Newton's equations of motions as differential equations are coupled in the following manner:

$$\frac{dx}{dt} = \dot{x}(t) = v(t), \tag{4.23}$$

$$\frac{dv}{dt} = \ddot{x}(t) = \dot{v}(t) = a(t), \tag{4.24}$$

where $a(t) = a(x(t), v(t), t)$. And since $F = ma$,

$$\sum_{i=0}^{N} F_i(t) = ma(t) = m\dot{v}(t) = m\ddot{x}(t). \tag{4.25}$$

If I consider Δt to be the time interval between successive sample boundaries of the discrete equation solutions, then a_n, v_n, x_n are the discrete solutions to acceleration, velocity and displacement at time $t_n = t_0 + n\Delta t$, where

$$a_n = \frac{F(x_n, v_n, t_n)}{m}. \tag{4.26}$$

The goal of any finite difference method is to determine the approximate value for x_{n+1} and v_{n+1} at time $t_{n+1} = t_n + \Delta t$. Finite difference methods converge on the actual solution to a differential equation as Δt approaches zero.

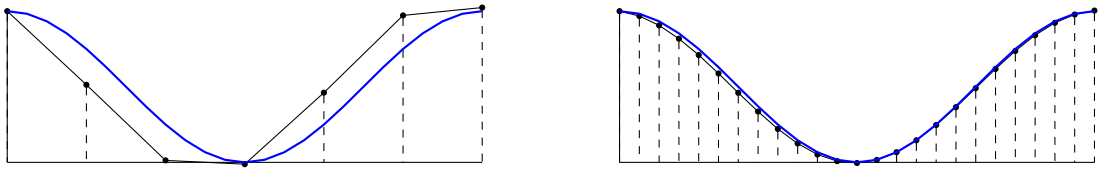

Figure 4.1: Numerical approximations to a continuous function with a large timestep (left) and a small timestep (right).

4.2.1 Integration Types

Finite difference methods [15] can be divided into two broad categories; multi-step methods which use several previous state approximations to calculate future states and Runge-Kutta methods which use several future calculations to approximate a weighted averaged future state. Of these categories, there are two approaches to deriving solutions; explicitly using a "forward" method and implicitly using a "backward" method. Explicit methods calculate a future state by calculating current state:

$$Y(t + \Delta t) = F\left(Y(t)\right). \tag{4.27}$$

Implicit methods solve by involving both the current and future state:

$$G\left(Y(t), Y\left(t + \Delta t\right)\right) = 0. \tag{4.28}$$

Backward methods often require root-finding algorithms to converge towards solutions. A common approach to using implicit methods to solve time-domain equations is to use a paired "predictor/corrector" algorithm. The predictor portion derives a rough approximation to future state from current state while the corrector portion refines the approximation iteratively. Backward methods that are adjusted using predictor-correctors are known as "semi-implicit".

4.2.2 Discretization Error, Accuracy and Stability

When calculating approximations using a finite difference method, there will be a difference between the value of the approximation and the exact actual solution's value. This error difference is known as discretization or truncation error

[10, pg. 639-640]. This error at any given point is known as local truncation error. The error compounded over the domain of the integration method is known as global truncation error. Often the global error is notated as the nth order of Δt, $\mathcal{O}(\Delta t^n)$. The higher the order, the lower global truncation error and hence, higher accuracy of the method. [1]

The stability of a method is measured by how well the method dampens errors and conserves energy. Differential equations that are "stiff" are those with rapid variation in the solution. Methods that react poorly to this high-frequency fluctuation often incrementally over or under-approximate the solution, resulting in exponential energy growth for over-approximation and exponential decay for under-compensation. For audio applications, it is often more desirable for the engine to "undershoot" a solution resulting in artificial damping rather than unbounded energy growth. For this reason, the engine favors approximations that tend to the latter case. It is important to choose a relatively small Δt to generate a stable solution (depicted in Fig. 4.1).

4.2.3 Explicit Integration Methods

The following is an overview of several well-known explicit methods and their respective strengths and weaknesses. I begin by consider the following Taylor series expansion of the equations of motion [12, pg. 492-493]:

$$v_{n+1} = v_n + a_n \Delta t + \mathcal{O}(\Delta t^2), \tag{4.29}$$

$$x_{n+1} = x_n + v_n \Delta t + \frac{1}{2} a_n \Delta t^2 + \mathcal{O}(\Delta t^3). \tag{4.30}$$

Euler Method

Euler's method is the most elementary forward numerical method [10, pg. 638].

$$v_{n+1} = v_n + a_n \Delta t, \tag{4.31}$$

$$x_{n+1} = x_n + v_n \Delta t. \tag{4.32}$$

[1] It is important to note that round-off error, which results from the finite precision on computers, also leads to error, but is much less significant than truncation error.

As can be observed, it is equivalent to the Taylor series expansion. It uses information only from the beginning of current state's interval to approximate the future state. Its local truncation error is $\mathcal{O}(\Delta t^2)$ while its global truncation error is $\mathcal{O}(\Delta t)$. For this reason, it is known as a first-order method, often producing unstable and inaccurate solutions.

Euler-Cromer Method

I can improve Euler's stability for oscillatory systems simply by using a last-point approximation.

$$v_{n+1} = v_n + a_n \Delta t, \tag{4.33}$$

$$x_{n+1} = x_n + v_{n+1} \Delta t. \tag{4.34}$$

This slight alteration, known as Euler-Cromer [15], conserves energy for most low-stiffness oscillatory equations, rendering this method significantly more stable and accurate than Euler.

Leapfrog Method

Another improvement of Euler's method is to use the middle velocity, instead of the beginning or ending velocity. This half-step method is known as "Leapfrog" integration [15].

$$v_{n+\frac{1}{2}} = v_{n-\frac{1}{2}} + a_n \Delta t, \tag{4.35}$$

$$x_{n+1} = x_n + v_{n+\frac{1}{2}} \Delta t. \tag{4.36}$$

Its important to note that this method is not self-starting, therefore the initial mean velocity must be calculated using

$$v_{\frac{1}{2}} = v_0 + \frac{1}{2} a_0 \Delta t. \tag{4.37}$$

This method provides more stability that Euler and is a commonly-used higher-order solver.

Euler-Richardson Method

I can improve the Leapfrog method for velocity-dependent systems by combining it with an Euler step. The Euler-Richardson is a method that combines these two methods [15].

$$\tilde{v}_{n+\frac{1}{2}} = v_n + \frac{1}{2}a_n\Delta t, \tag{4.38}$$

$$\tilde{x}_{n+\frac{1}{2}} = x_n + \frac{1}{2}v_n\Delta t, \tag{4.39}$$

$$\tilde{a}_{n+\frac{1}{2}} = \frac{F(\tilde{x}_{n+\frac{1}{2}}, \tilde{v}_{n+\frac{1}{2}}, t_{n+\frac{1}{2}})}{m}, \tag{4.40}$$

$$v_{n+1} = v_n + \tilde{a}_{n+\frac{1}{2}}\Delta t, \tag{4.41}$$

$$x_{n+1} = x_n + \tilde{v}_{n+\frac{1}{2}}\Delta t. \tag{4.42}$$

Note that two acceleration values are needed, the beginning acceleration and the middle acceleration. I want to compute as few acceleration terms as possible in the engine. Luckily, Euler-Richardson is significantly more accurate than Euler, so the time step can be much larger, reducing its computational footprint. This method is particularly excellent at accurately approximating oscillatory solutions, often compute solutions only a few bits different. However, its stability range is much smaller than even Euler-Cromer, so it does not fair well against moderately stiff systems.

Verlet Method

A popular choice of molecular simulations, Verlet's method [10, pg. 640-642] is a second-order, explicit central difference method that uses the current position, previous position, and current acceleration.

$$x_{n+1} = 2x_n - x_{n-1} + a_n\Delta t^2 + \mathcal{O}(\Delta t^2). \tag{4.43}$$

This method is equally stable as Euler-Cromer, but without needing to compute velocity directly. For systems that do not depend on velocity, this is a particularly efficient method. However, for systems that depend on velocity, one must directly compute velocity from positions:

$$v_n = \frac{x_{n+1} - x_{n-1}}{2\Delta t}. \tag{4.44}$$

This is a first-order approximation, which is very inaccurate. Additionally, when using very small time steps (as I find in the engine), there is a significant amount of round-off error, resulting in the method often reporting zero velocity for any values below a fairly large epsilon.

Velocity Verlet Method

In order to avoid the velocity pitfalls of Verlet but still gain its computational advantages, one can use the Velocity Verlet form [10, pg. 641-642]

$$x_{n+1} = x_n + v_n \Delta t + \frac{1}{2} a_n \Delta t^2, \tag{4.45}$$

$$v_{n+1} = v_n + \frac{1}{2} \left(a_{n+1} + a_n \right) \Delta t, \tag{4.46}$$

It is a second-order method for both position and velocity. Alternatively, one can compute velocity from current position, future position and future acceleration (storing this value to be used for the next position update),

$$v_{n+1} = \frac{x_{n+1} - x_n}{\Delta t} + \frac{1}{2} a_{n+1} \Delta t. \tag{4.47}$$

Beeman Method

An improvement on Verlet is Beeman's method [3], which uses previous, current and future acceleration terms,

$$x_{n+1} = x_n + v_n \Delta t + \frac{1}{6} \left(4a_n - a_{n-1} \right) \Delta t^2, \tag{4.48}$$

$$v_{n+1} = v_n + \frac{1}{6} \left(2a_{n+1} + 5a_n - a_{n-1} \right) \Delta t. \tag{4.49}$$

$$\tag{4.50}$$

Alternatively, velocity can be derived from slightly different weights,

$$v_{n+1} = v_n + \frac{1}{12} \left(5a_{n+1} + 8a_n - a_{n-1} \right) \Delta t. \tag{4.51}$$

It improves Verlet in terms of energy conservation, providing significantly more stability. It does not improve on Verlet's accuracy, however.

Runge-Kutta

A commonly used, but often inefficient method is the classical Runge-Kutta (or RK4) [12, pg. 493-495]. This method computes a weighted average of the iterative approximations from the current state.

$$x_{n+1} = x_n + v_n \Delta t, \tag{4.52}$$

$$v_{n+1} = v_n + \frac{(k_1 + 2k_2 + 2k_3 + k_4)}{6} \Delta t, \tag{4.53}$$

where

$$k_1 = a\left(x_n, v_n, t_n\right), \tag{4.54}$$

$$k_2 = a\left(x_n + \frac{k_1}{2}\Delta t^2, v_n + \frac{k_1}{2}\Delta t, t_n + \frac{\Delta t}{2}\right), \tag{4.55}$$

$$k_3 = a\left(x_n + \frac{k_2}{2}\Delta t^2, v_n + \frac{k_2}{2}\Delta t, t_n + \frac{\Delta t}{2}\right), \tag{4.56}$$

$$k_4 = a\left(x_n + k_3 \Delta t^2, v_n + k_3 \Delta t, t_n + \Delta t\right). \tag{4.57}$$

RK4 does indeed provide significantly more stability and accuracy than Euler. However, it should also be clear that RK4 requires the times as many acceleration computations. For particle systems and systems where acceleration has many dependent variables, it is still more practical to use an Euler or Verlet method. The computational cost of RK4 is greater than using an Euler or Verlet variation at one-fourth the time step.

4.2.4 Numerical Methods Conclusions

In summary, the engine requires a fixed time step (at the audio sampling rate) and demands energy conservation (stability) over accuracy. It also needs to be able to provide this stability for as little computational expense as possible. Therefore, the engine uses a modified Beeman's algorithm, to benefit from higher stability for relatively low computational cost. Using Beeman's, the engine computes acceleration terms once per sample and integration is optimized by storing previous acceleration terms as precooked variables.

Chapter 5

Implementation Details

5.1 Low-Level Architecture

As stated previously, in addition to the requirements for sufficient accuracy and stability, the engine demands exceptionally high real-time performance to ensure uninterrupted stream rendering. In order to achieve this high level of performance, every aspect of the low-level engine must be profiled and optimized to achieve maximum throughput.

The major difficulty in achieving high-performance comes from the shear amount of computations that must be performed. For every calculated audio sample, the engine must calculate the sum of all forces on every particle, integrate each particle and then measure the energy for every particle for each listening point (see previous chapter for details). For small numbers of particles and a sparse network of springs, this is feasible, but as the order of particles and the density of the network increases, the computational load goes up exponentially. My optimization strategy deals directly with parallelizing computations, through the use of SSE registers and thread concurrency, and reducing cache misses by optimizing the engine's data structures.

Once an acceptable speed have been reached, the next order of business is tackling the state management of the engine's primitives and how it communicates with the outside world about its internal state, including the creation, deletion, modification and querying of the engine's primitives. I deal with this issue by

implementing a reference management system to manage external requests and a network of associative arrays to manage internal state.

5.1.1 Data Structures

Engine primitives are packed in carefully organized data structures that are designed to improve cache hits and reduce redundant calculations. The engine treats particles as one data type and springs/dampers as a another data type.

Precooked Coefficients

Condensing relatively-static variables into fewer run-time operations can significantly reducing per-sample computation during integration. Take for example the following classical integration of a mass on a spring:

$$F_n = -kx_n, \tag{5.1}$$

$$\ddot{x}_n = \frac{F_n}{m}, \tag{5.2}$$

$$\dot{x}_{n+1} = \dot{x}_n + \ddot{x}_n \Delta t, \tag{5.3}$$

$$x_{n+1} = x_n + \dot{x}_n \Delta t, \tag{5.4}$$

which is a total of 6 operations per particle per sample. If it can be assumed that Δt is fixed (and it will most certainly be under most audio-specific conditions), the computation costs can be reduced by two-thirds:

$$g_n = Bx_n, \tag{5.5}$$

$$x_{n+1} = x_n + g_n, \tag{5.6}$$

where $g_n = \dot{x}\Delta t = \ddot{x}\Delta t^2$ and $B = \Delta t^2 \frac{k}{m}$, reducing the calculation to only 2 operations per particle per sample.

Warehouse

Both particles and springs have dependent variables (such as mass and stiffness constants) that they require when calculating their precooked coefficients.

These values are not needed during integration and only to be referenced when updating or calculating the engine primitives' properties. These variables are stored in data warehouse structures in non-contiguous memory.

Reverse-Stack Structures

These two types are placed in tightly packed array in contiguous memory. I will use two data structures for this; stacks and reverse-stacks.

Reverse-stacks are simple stack structures that push to the bottom instead of the top of the structure, meaning the first entry exists at the end of the structure's memory location, the second item exists right before the end, and so forth as illustrated:

$$data[i] = mem[(N-1) - i], \tag{5.7}$$

where i is the lookup index, N is the maximum size of the structure, mem is the location in memory and $data$ is the desired item requested.

Heap Allocation

First I allocate a large heap that I will divide between particles and springs. At the front of the heap, I place a reverse-stack structure of springs, which reserves half of the heap. At the middle of the heap, following the end position of the reverse-stack, I reserve a stack structure of particles. This formation allows us to have very large maximum sizes for particles and springs while at the same time only submitting contiguous memory to the cache.

Process Concurrency

As stated previously, integration involves four ordered stages (force calculations, collision detection, state integration, and constraint calculations). Though the entire process must be ordered, the individual stages are not. Therefore, the engine can benefit from asynchronously completing each stage. In order to accomplish this, the engine initially deploys worker threads that breakup the tasks for each stage. The threads block until they all have completed a given stage at which point they continue.

SIMD Intrinsics

During integration, I take advantage of modern vector instrinsics for all particle and spring calculations. During the force calculation stage, I integrate four springs at a time in order to also vectorize scalar operations for the springs. I also integrate four particles at a time for similar reasons.

5.1.2 State Management

The internal state management of the engine is similar to the structure of a database management system, except that the low-level structures and operations are optimized for real-time application. By managing state internally, many powerful higher level features can be exposed to user for a wide range of applications.

Unique Keys

The engine maintains state internally, rearranging structures in order to optimize cache throughput. It is desired to not expose its internal structure to the user since the memory locations of any given data may not persist from update to update. For this reason, the engine provides users with unique handles in order to access engine primitives. Each particular instance of a primitive has a unique handle and handles are not reused. The reason for this is that both the user and the engine can remove primitives. The user may decide to remove a primitive through the editing interface, while the engine may decide to destroy a given particle or spring as a result of its update routine (via rupturing). Each unique handle is then resolved internally to the corresponding primitive type and memory location in order to operate on it as requested.

Associative Arrays

The engine also internally maps out the connections between particles through associative arrays. There are two associative arrays, one for particles and one for springs. The associative arrays are arrays of lists. The particle associative array lists all the springs connected to each particle, the spring associative

array lists the two particles connected to each spring. The engine uses these arrays to determine the network topology of the system. For example, users can query the engine to report all particles within a network and their depth from a given particle.

Queries

In addition to queries regarding network topology, the engine handles queries that can filter for specific parameters, much like a SQL database. Queries returning all particles within a given radius or all particles accelerating beyond a given threshold are two examples. This information can be used to allow Users to develop custom force functions or other routines that can be attached to the core engine. Queries can provide user-defined functions for comparisons, allowing users to optimize search results. Queries on primary primitive structures (i.e. structures submitted during integration) are generally much faster than secondary queries (i.e. on warehouses).

State Preservation

State Preservation is important for many purposes. Since the models are dynamic, a given snapshot of the model may produce significantly different results than another snapshot of the same model. States allow users to store and recall particular configurations of a given model. This is especially useful in order to recover after an instability. A particularly important state is the "initial" state of the model, which the engine treats as the default configuration to save and restore the model in. The "initial" or resting state of the model is the ideal state of equilibrium across the entire model. The engine can attempt to recover any stable configuration back to the "initial" state by increasing air drag significantly, measuring the total energy of the model and lowering air drag as energy drops, until the model reaches a static, stable configuration.

Instancing

In addition to model state preservation, the engine can preserve and modify a sub-state of the model and instance another copy with altered properties. This instancing technique allows the engine to render many pitched copies of a given model. The pitch is estimated with a eigenvalue decomposition and the model's force matrix is adjusted to attain a desired fundamental frequency. Instancing is general enough to be applied to other parameters other than pitch, such as damping or friction as well. This technique can also benefit from only integrating instances that have been driven by some external force, allowing a much wider arsenal that if the entire model was always required to be rendered whole.

5.2 High-Level Interface

The high-level interface is an editing and performance application that provides real-time audio-video I/O to the engine. Though there is a distinction made between editor and instrument, the engine does not distinguish the two; modification, creation and deletion of primitives are treated as instantaneous non-linearities in the ODE system.

Once stability can be reasonably assured, I can move on to developing a high-level user interface. Before I continue detailing the system's interface design, I shall examine some applied theory of user interfaces. It should be noted that the following theoretical observations come after the fact, as a way to describe how one comes to interact with a system, and not the other way around.

5.2.1 Interface Theory

Interfaces represent the relationship between user and data. I fulfill this relationship through symbols, which represent the underlying data. Symbols can represent the data with as little or as much detail as desired, but whatever the case, using symbols that are unambiguous, concise, and consistent is paramount to efficient and intuitive interface design. These symbols are generated from the data and when the data changes, the symbolic representation reflects those changes.

Control and Feedback

The feedback between the symbol and the data allows the user to understand the basic structure of the interface, but by interacting with the system, one learns a great deal more; most significantly, one subconsciously prototypes its behavior, which leads the user's to learning its control apparatus. Efficient control is measured by the amount of power given to the user by the amount of effort required for that power. Control is guided by feedback, and when there is consistent analog between human gesture and directive, the user will grasp the underlying relationship. The simpler and more direct the analog, the quicker the average user will learn the mechanic and be able to address higher-order problems. With a suitable balance between power and effort, the user can make subtle and dynamic changes to the underlying data, which in turn is reflected by the symbolic representation.

Mastery

In order for the user's to master control of the system, several significant leaps of understanding must take place. Initially, after observing the presented interface, one seeks confirmation of the meanings of the symbolic operations. After this, one interacts with the interface and observes the variations in the symbolic representation, produced by the user's changing of underlying data. Once one perceives the correlation between data changes and symbol variation, one begins to recognize behaviors associated with changes one imparts on the system. When the representation is updated at a rate where motion is perceived, one is given the opportunity to directly observe in time the symbol-to-data correlation. With this level of understanding, one learns to align the user's control apparatus, balancing the user's effort with the power one needs. Once one successfully learns this balance, one masters the control, and regulates it to a background process. Once control is out of the foreground, one can then focus the user's mental prowess on higher-order operations, namely building and playing musical instruments.

5.2.2 Model-View-Controller

The High-Level Application adopts the model-view-controller (MVC) design pattern [9, pg. 4-6]. Care is taken to ensure the interface remains as modular as possible. The low-level engine is built specifically to be highly modular; the view and controller components of the application are designed to operate in a similarly agnostic fashion.

5.2.3 Graphics Engine

The application hosts a modern OpenGL graphics rendering engine. The engine supports modern techniques for shading and lighting effects. Though these can be used in artistic presentations, they can assist users in determining the depth of the model (using ambient occlusion methods) as well as the mesh normals (using diffuse and specular lighting methods). The graphics pipeline involves rendering ray-traced spheres and cylinders to represent the particles and springs. Ray-traced primitives involve projecting simple geometry onto a camera-aligned billboard. This technique produces the effect of perfectly smooth meshes with accurate depth projection and orthogonal vectors for each pixel on screen. This allows us to follow up the ray-tracing technique with a Phong lighting model for ambient, diffuse and specular lighting. Following lighting calculations, I use a full-screen post process; a screen-space ambient occlusion algorithm that accounts for occluded light when objects persist near enough to one another. Lighting effects allow users to have a strong sense of the topology of their model. The ambient occlusion significantly improves this sense of topology, and is common in many molecular rendering programs.

5.2.4 The Editing Interface

A significant issue with this proposed system is how it should allow users to construct and modify their virtual instruments. It can be a tricky situation, since users will desire both the conventions of synthesizer programs as well as 3-D modeling programs. Fundamentally, the proposed system's editing interface should

provide users with workspace to construct their models with as little interference as possible. Conventions developed in 2-D and 3-D artistic modeling programs are considered. Though the editor does not need the precise sophistication of CAD software, some of the techniques developed for use in CAD editing software can be adopted in this case.

At the system's lowest level exists individual particles and the spring forces that act on them. These are the elements that comprise the ODE system. Particles have physical state as well as several associated properties such as radius and mass. Springs generate forces which describe how particles interact. They have associated properties such as tension and damping values. From these simple elements and a handful of properties, I can produce a huge variety of sounds in this environment. Many models begin to sound satisfyingly complex around several tens of elements, but other models may require closer to several hundred to a few thousand particles to sound equally convincing. Such a scenario may seem quite unwieldy. Therefore, I strike a balance between power and effort by introducing the notions of prefabrications and presets, on top of primitives.

Prefabrications can be seen as groups of particles and their related spring forces. They are practical for constructing instruments without having to constantly operate at the tedious elemental level. Presets can be seen as prefabrications with predefined values and value ranges. The point of this object hierarchy is to enable the user to balance their amount of power with respect to the effort involved for that power. Constructing everything solely at the atomic level is democratic but also inefficient, especially when the user creates a simple string out of particles and then wishes to use many more strings to complete the instrument. A prefabricated string can make this task much easier. Prefabricated strings can also have presets that define different parameter vectors for various kinds of strings (rope preset, metal preset, rubber preset, etc). By implementing prefabrications and presets, the user can work with fewer, more complex elements instead of the often-overwhelming amount of individual mass and spring elements.

5.2.5 The Performance Interface

Performance tools are the most complex group of functions. Some tools simply move the particles around. Others inject energy at localized particles or particles within a given radius. Some common techniques include plucking, hammering, blowing, scrapping, pulling, bowing, bouncing, tapping, and shaking; all of which can modeled by various ways of injecting energy into the system. Regardless of the details of their implementations, they are all different approaches to system excitation. Some tools are controlled via the mouse; but there is no reason to exclude the range of human interface devices common on modern computers. Keyboards, joysticks, microphones, external MIDI controllers, custom serial devices, and network clients are some a few of the possible ways to control various excitation methods. The tools should be generic enough to attempt to accommodate a range of playing styles but also offer some specialty tools such as:

- A heat gun that slowly burns the instrument's strings, loosening and unraveling them until they snap apart.
- A electromagnetic tool that pulls and pushes particles surrounding it at various frequencies and amplitudes.
- Simulated water that fills a portion of the screen, respectively changing the pitch of any part of the instrument that is underwater.
- A glue gun that sticks elements together, binding them with a rigid spring.
- A "kelvin" gun that works as the opposite to the heat gun, gradually increasing the rigidness of the structure.

Chapter 6

Applications

This chapter hopes to illustrate the practical applications of the proposed engine. Previously, the reader has been introduced to the philosophical and technical background to the realized audio engine in its current state. *Ruratae* is the prototype physics-based audio engine. Ruratae is the Mycenaean Greek word for lyrists, those that play the lyre. It seems suitable, since the project takes much from Greco musical pioneers and those they inspired (e.g. Harry Partch).

Ruratae, in its present form, is a manifestation of many of the techniques discussed in this document. It outlines the possibilities of dynamic, real-time virtual musical instrument interactions, as well as points out many of the emergent interface and design issues of the techniques. Users can create, modify, strike, pull and destroy models with keyboard and mouse input. They can also save model state, allowing them to preserve a model in filespace or allow them to "undo" changes they make to the model. There is also a "music-box" that loads a pre-rendered state that includes several instruments that I've created for users to experiment with. This music-box acts as a tutorial introduction to the system. Its strongest point is that is allows users of little-to-no musical background to immediately begin creating, experimenting and exploring physical interactions with their virtual musical instruments. The software is entirely self-contained, requires no additional configuration by the user, and almost any modern computer will satisfy its run-time requirements. There are certainly many areas that can be improved on in the software.

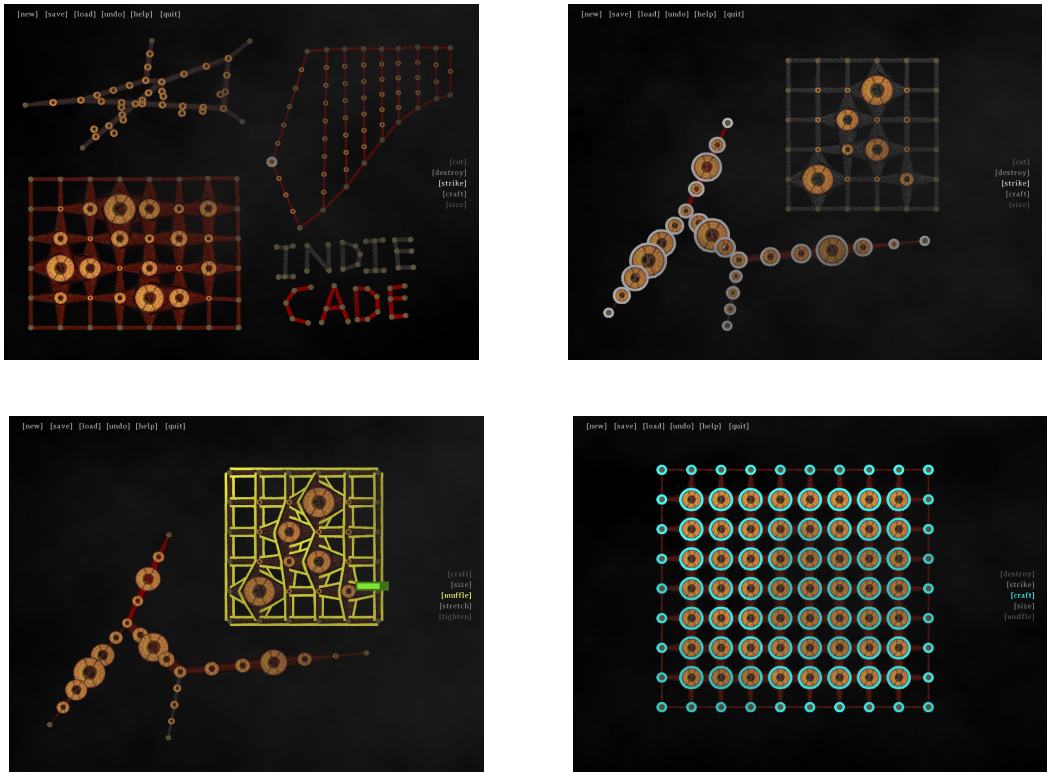

Figure 6.1: Screenshots of Ruratae v0.1b

6.1 Software Overview

The software (screenshots shown in fig. 6.1) is written in C++, and only uses three external libraries; OpenGL for graphics rendering, PortAudio for cross-platform audio driver access and GLFW for cross-platform windowing and system event management. As noted in the previous chapter, there was considerable attention paid to optimizing the physics calculations in order to make the system very responsive. There are options to save and load model state. The software presents the user with a canvas on which to create two-dimensional instruments. There are a selection of input modes: craft, size, muffle, stretch, tighten, strike, cut, and destroy. Each of these modes correspond to distinct types of interactions with the system. Some modes interact only with masses and others interact with springs, some modes interact with both primitive types.

Craft mode is the primary construction mode of the system. Crafting allows the user to create strings, plates, and connections between structures. Using the

mouse, the user can craft strings of masses connected by springs by clicking and dragging. The start and end positions of the string are determined by the location of the cursor when the mouse button goes down and goes back up. By holding the shift button, the mode shifts to crafting a rectangular plate, with the mouse-down and mouse-up positions determining the opposite corners of the plate. The user can also connect strings to pre-existing masses by moving the mouse over them when pressing or releasing the mouse button.

Size mode allows users to modify the mass value of the particles. The interface allows users to slide between a range of pre-determined values while the mouse button is held down on a mass. If the user holds shift, all masses are selected that are connected via springs to the mass under the cursor. The range of values is limited to a range known to be stable for the integration method chosen. By changing the size of the masses, the model's pitch changes accordingly, in a determinant manner.

Muffle mode allows users to modify the damping value of the springs. A similar slider interface is designed for selecting a damping value. The user can also hold shift to select the connected network of springs. The damping value produces a wide range of effects; because damping only occurs between individual springs, the behavior of the system can be very dynamic in the case that some sections of the model are very muffled and other sections are not, this produces a wide range of possible spectral compositions.

Stretch mode applies to the resting length of the springs. The stretch values are between zero and one and set the resting length as a percentage of the initial length of a spring between two masses. Holding shift affects a network of springs. This approach to setting resting length is useful for introducing the concept of rigid springs to users, but in hindsight, it is clear that a more formal approach for allowing variable rest-lengths not dependent on initial distance would be more beneficial. The non-linear chaotic motion of springs with resting lengths is a very interesting phenomena. Allowing the user the ability to explore the phenomena at will was considered a particularly effective demonstration of the benefits of this form of physical modeling.

Tighten mode is very similar to Size mode, except that it affects the stiffness constants of springs instead of the mass value of particles. It essentially has a similar effect on the model, but the actual normal modes vary differently depending on the model's configuration. Like the previous modification modes, this mode uses a slider that is enacted when the user mouses down on a desired spring. Holding shift affects a network of springs.

Strike mode enables a simulated striking of the model. This is done by introducing an impulse vector into the system. The magnitude and direction of the impulse are randomly varied, as an attempt to reproduce microvariation of attack in regular performance. This randomization feature was included because of the significant limitations of keyboard-mouse performance. When more sophisticated devices are introduced to the system that can account for actual microvariation in performance, this randomization will not be as important. In the meantime, the randomization significantly improves the realistic effect of striking the system. What is particularly interesting is that the listening position of the user is determined by the mouse position, meaning that the user always listens to the model from the perspective of the striking point. This allows the changes in the model's modal amplitudes to appear much more drastically, resulting in a hyper-realism of the model's vibration.

Cut mode allows the user to "cut" springs as if they have physical dimensions. This effectively breaks the bond between particles. This break will cause a discontinuity in the system's equilibrium, resulting in a modal reconfiguration. The new distribution of forces will impulse the network and depending on the stiffness of the system, will inject a rather violent burst of energy into the system. Without significant damping applied to the system, it can very often become unstable. *Destroy* mode is very similar to "cut" mode, except that it affects particles. It should be noted that systems where spring resting length equals actual equilibrium state, the energy injected will be canceled out immediately, and no model reconfiguration will occur.

Figure 6.2: A blank scene in Ruratae

6.2 Getting started with Ruratae

In this section, I will guide the reader through the process of creating, modifying, playing and destroying a simple instrument inside the Ruratae demo software.

6.2.1 Crafting the string

We will begin by loading the software, which initializes a blank scene (depicted in Fig. 6.2). The menus located on the right-side panel represent a list of the various editing and performance modes of the software. Select "craft" mode from the list to create a string of particles connected in a line by springs. To begin, position the mouse near a corner of the screen, click the left mouse button and drag the mouse across the screen. You will see a faint overlay depicting the string it will create once the left button is released (depicted in Fig. 6.3).

6.2.2 Modifying its properties

Once an object is created, it is immediately being simulated, which means it can be performed on or modified in real-time. A string defaults to having

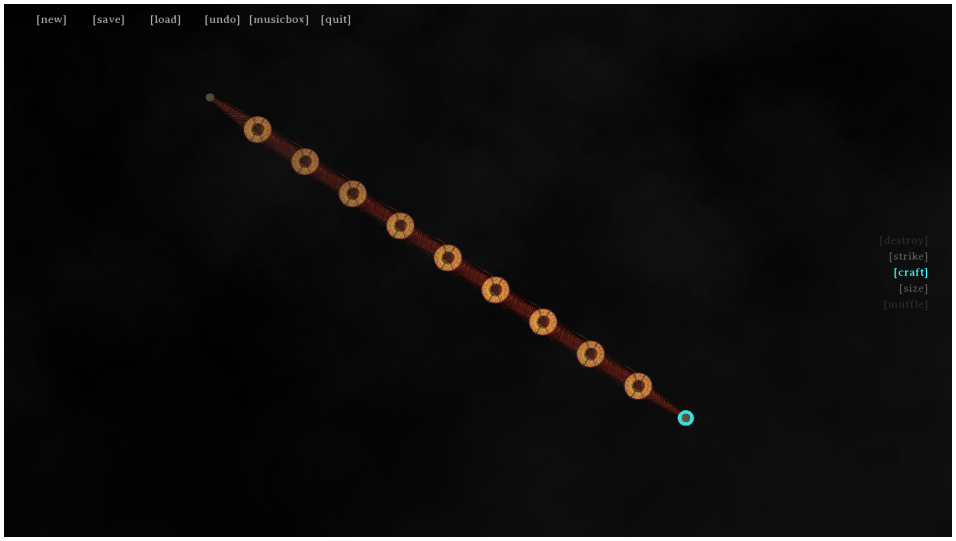

Figure 6.3: Dragging the mouse along to make a string

homogenous mass and spring constants across its area. To create a string that is less homogenous, we will modify some of the particles' mass values. Select "size" mode from the list to be able to adjust mass values for particles. Move the mouse over a particle and click the left mouse button. A small slider widget will pop up above the particle, indicating its current mass value. While holding the left mouse button down, you can move the mouse to the left or right to adjust the value (depicted in Fig. 6.4). If the slider value is moved all the way to the left, the mass becomes a "fixed" mass, or "wall of infinite mass". One can observe that the ends of the string are both "fixed"-mass particles. Adjust several particles' mass values to make the string fairly heterogenously dense.

We can also adjust properties of the string's springs. Select "stretch" mode from the list to adjust the springs' resting lengths. Move the mouse over a spring and click the left mouse button. As with "size" mode, a slider pops up indicated the available range of values to select (depicted in Fig. 6.5). Adjust some of the springs' resting lengths to make resting length fairly irregular over the string.

Likewise, we can adjust the damping constant for springs in the string in almost the same way. Select "muffle" mode and adjust values on springs in the same way as for "stretch" mode. You can hold down *shift* while holding down the left mouse button to muffle a group (depicted in Fig. 6.6). One can also select

Figure 6.4: Modifying a particle's mass

Figure 6.5: Modifying a spring's resting length

Figure 6.6: Modifying all springs' muffle/damping qualities

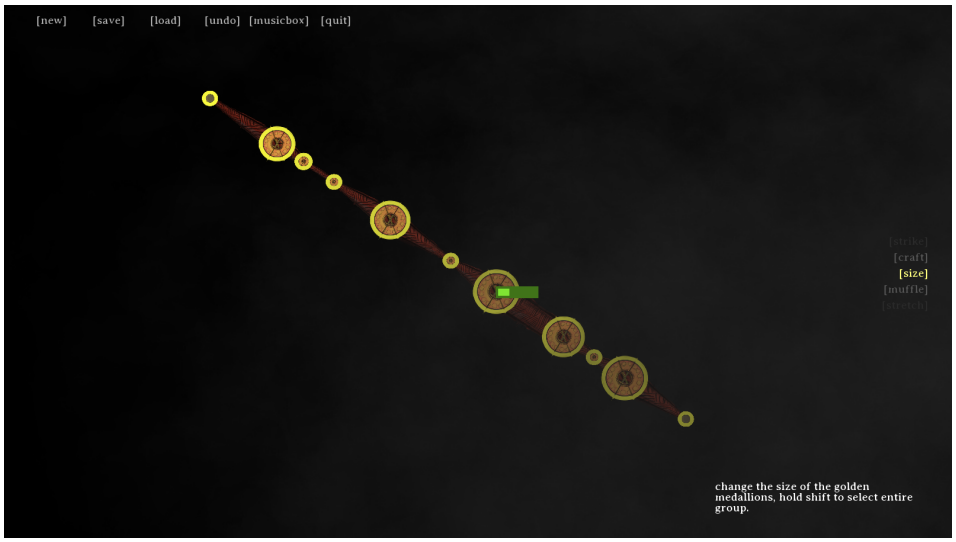

Figure 6.7: Modifying all particles' masses

Figure 6.8: Striking the string

a group of particles by holding *shift* while holding down the left mouse button (depicted in Fig. 6.7).

6.2.3 Striking the string

Now that our model is complete, we can perform on it in various ways. Two excitation methods that exist in this demo are striking and plucking. We can strike the string by selecting the "strike" mode. Strike the string in various locations to hear the string's resonant frequencies being excited in different proportions (depicted in Fig. 6.8). Plucking the string involves grabbing a particle along the string, displacing it and then releasing it (depicted in Fig. 6.9).

6.2.4 Cutting the string and destroying its end-points

We can also perform on the string by cutting it. To cut a spring between two particles, select "cut" mode and position the mouse over the spring desired to be cut (depicted in Fig. 6.10). When you click on the spring, it will be removed and the string will split into two parts, which each part rushing away from the cut (depicted in Fig. 6.11). By selecting "destroy" mode, you can delete particles instead of springs (depicted in Fig. 6.12).

Figure 6.9: Plucking the string

Figure 6.10: Getting ready to cut the string

Figure 6.11: After cutting the string, resulting in two independent string instruments

Figure 6.12: After destroying several particles

Figure 6.13: Reconnecting the pieces together

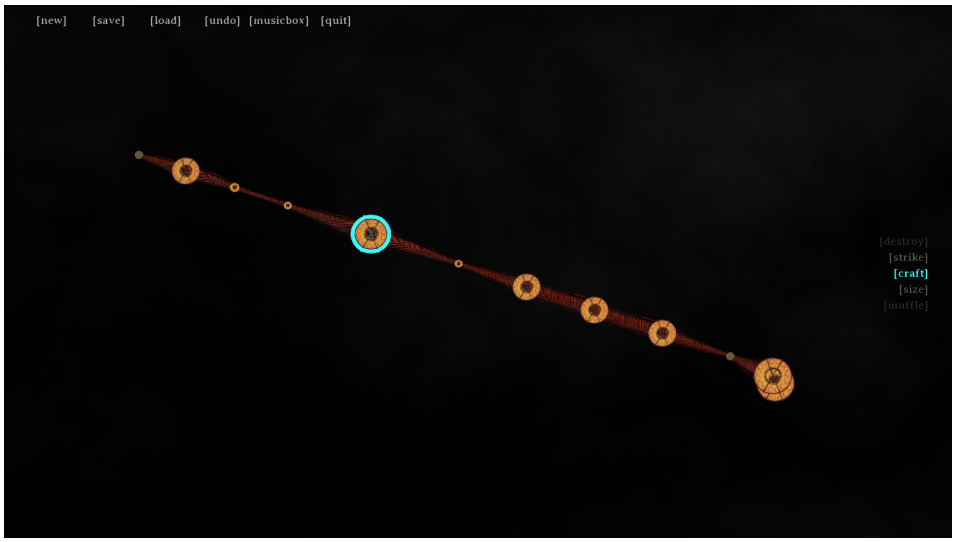

Figure 6.14: After reconnecting, a new string is born

Figure 6.15: Selecting the entire string to be destroyed

At this point, we can reconnect the fragments of the previous string, to give birth to a new instrument. Select "craft" mode and position the mouse over a particle onscreen. Click the left mouse button and drag it along the screen until the mouse is positioned on top of a different particle from a different fragment (depicted in Fig. 6.13). Releasing the mouse button will create a string between the two particles. Behold, a new instrument is born (depicted in Fig. 6.14). Finally, we can clean-up our work space either by selecting "destroy" mode and holding *shift* to select the entire group of particles and destroying them (depicted in Fig. 6.15), or by clicked *[new]* at the top of the screen.

6.3 A Musical Taxonomy

As a demonstration of the variety of musical timbres produced by the system, I will examine three instruments created in the system (depicted in fig. 6.16) and explore their sonic possibilities when modified using the editor tools. The first will be a harp instrument, which is a collection of nine independently vibrating strings. The second will be a drum box, a two-dimensional "drum"-like membrane that will demonstrate how the properties of the system affect higher dimensional bodies. The third instrument is a non-realistic (but still physically-

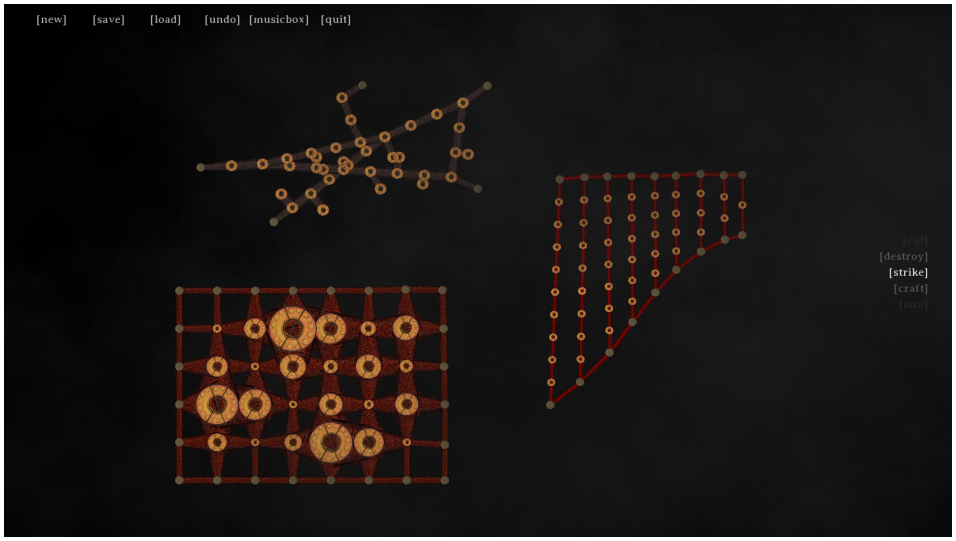

Figure 6.16: Three virtual instruments created in Ruratae

informed) instrument that is comprised of a few taut strings connected together, with noise-maker bodies attached along the strings. For this taxonomy, I have included several audio examples that demonstrating the various characteristics. As is required, these examples are available as a compressed archive included with the electronic version of this document and are also available from the archive in hard-copy form. It is important to emphasize that these are only some of the possible interactions with these simple instruments that can produce unique sonic results. Unfortunately, the actual dynamic and responsive elements of this system simply cannot be captured using prerecorded audio samples. It would be quite impossible to exhaust the system vocabulary in this format alone. However, I hope that these examples give some illustration of the system's potential power and versatility. For archival purposes, I have labeled the musical examples by name of the instrument, followed by a incremental number.

6.3.1 9-string harp

The 9-string harp (shown in fig. 6.17) is an object that, like its description states, has nine independent vibrating strings that are grouped together but not coupled. Each of the strings acts as a complete unit, the properties of all the

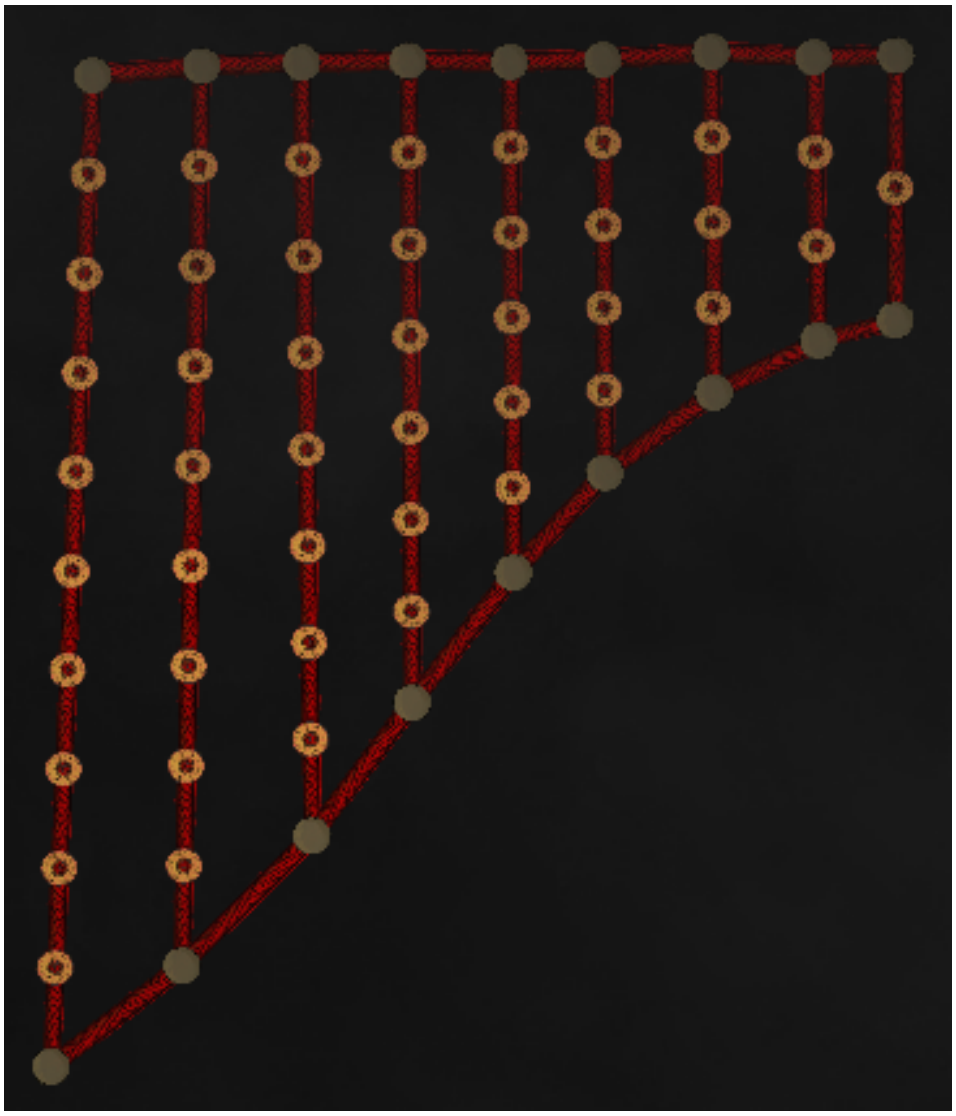

Figure 6.17: 9-string harp instrument modeled in Ruratae

harp's particles and springs are the same, the only difference between strings is the number of particles of which they consist. Harp Example 1 (Video 1) demonstrates plucking technique on each string, from the lowest to the highest. Plucking is produced by grabbing a particle, displacing it from its equilibrium, and releasing the particle. Harp Example 2 (Video 2) demonstrates striking of the strings, from the lowest to the highest. There are spectral differences between plucking and striking behaviors which result from exciting different modes in each case. Harp Example 3 (Video 3) demonstrates scraping of the strings, from the lowest to the highest, where a scrape moves from one end of the string to the other. A scrape is simply a rapidly-repeating (and slightly varied) striking along the body. As the scrape travels across the string, different tranverse and longitudinal modes are excited, producing a shift in spectral energy. Harp Example 4 (Video 4) demonstrates strumming up and down along all nine of the harp's strings. Even though this strumming behavior is simply generated by the mouse dragged along the object, it shows how responsive the system can be to gestural input from the user. Harp Example 5 (Video 5) demonstrates how the plucking behavior changes when the strings are slightly muffled. Harp Example 6 (Video 6) demonstrates how the plucking behavior changes when the strings are almost entirely deadened. Note the variety of sound qualities between these various dampened systems. The more deadened the system, the more hollow it appears. The more undamped the system, the more metallic it appears. It is important to note that the relationship between the modes produced in these examples depends both on system damping and the amount of energy injected through plucking. Harp Example 7 (Video 7) demonstrates how varying the rigidness of the spring lengths affects the system's modes. In this example, all nine of the harp's string are given more "rigidness" (i.e. the desired length of the strings' springs are increased) and then the reverse is applied. One can clearly hear the "popping" of the strings as they move from being at rest at their initial end points and then deformed into a rigid and out-blown state. Additionally, one can hear the reassembling of the strings as their rigidness is reduced, allowing them to vibrationally return to their initial resting positions. Harp Example 8 (Video 8) demonstrates the qualities of "rigid" strings,

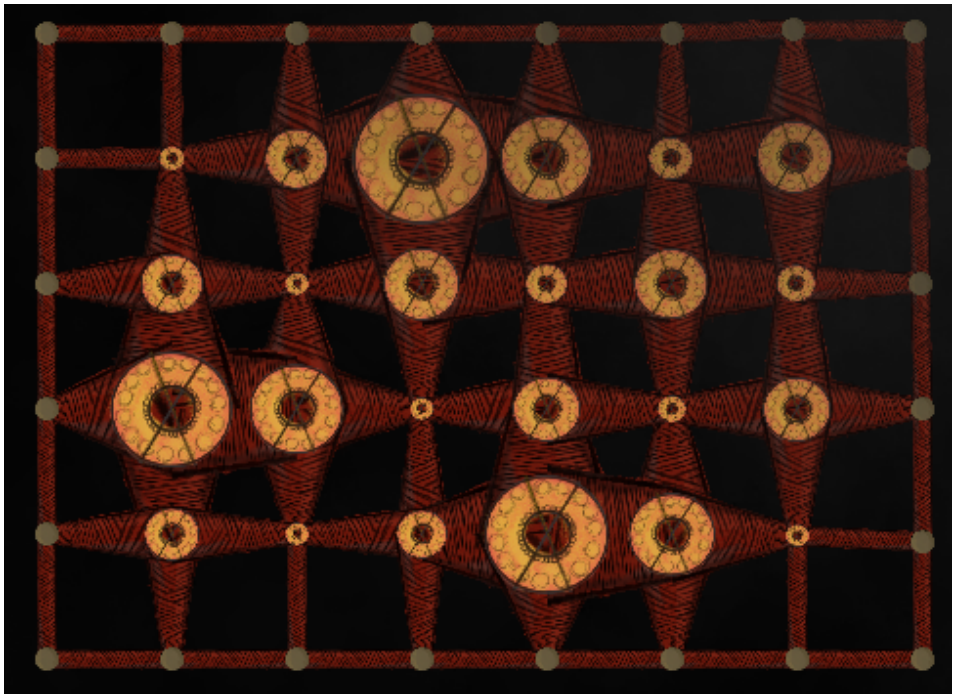

Figure 6.18: Two-dimensional drum membrane modeled in Ruratae

an example melody is played on the rigid strings. Note only the "rigidness" of the strings has been changed, but this still affects their pitch. Harp Example 9 (Video 9) strums up and down on the harp again, this time while the strings are still slightly rigid, producing a hollow, almost ceramic quality.

6.3.2 Two-dimensional drum membrane

The drum membrane object (shown in fig. 6.18) is assembled from particles connected by springs arranged in a grid. Each of the particles has a different mass, which produces various changes in impedance along the object, resulting in a drum-like, inharmonic spectrum. Drum Example 1 (Video 10) demonstrates striking around the object in a circular pattern. Drum Example 2 (Video 11) demonstrates scraping along the same path. In both of these examples, one can hear how the impedance changes across the object affect the modal composition of the sonic result. Depending on the horizontal and vertical position of the strike, this determines which longitudinal and transverse modes are present in the vi-

bration. Drum Example 3 (Video 12) plucks at various points along the object's shape. Its important to note that even though this object is considered to have a defined two-dimensional shape, the user can still deform it by grabbing and displacing particles (and subsequently, releasing them). Drum Example 4 (Video 13) demonstrates how muffling affects the vibration of the membrane. A large amount of damping is applied to the model and then it is scraped along its edges and through its center. Also, note the similarity in result as compared to individual strings with and without large amounts of damping. Drum Example 5 (Video 14) is a demonstrative melody played on a membrane with almost no muffling, producing a very resonant ringing that lasts for a long duration. The object in this state produces complex spectra comparable to that of a medium-sized tam-tam. This example melody demonstrates the musicality of suitably complex models. Drum Example 6 (Video 15) informs the listener of the sonic result of shaking the drum membrane with fairly rigid springs. A shake is producing by grabbing a particle in the model and then rapidly moving that grabbed particle around, causing the system to (sometimes violent) re-stabilize itself at every thrust. With sufficiently rigid springs, this produces an audible sloshing sound, which results from the non-linear resonances of springs with non-zero resting lengths. Drum Example 7 (Video 16) increases the rigidness of the system and also tightens the springs (increases the spring constants), producing a crackling sound reminiscent of crunching an aluminum can. Drum Example 8 (Video 17) demonstrates how responsive the modal composition is to the location of scrapes along a homogeneous membrane (one where all particles have roughly the same mass). Note that this example's membrane has a moderate amount of rigidness, which adds to the hollow, or ceramic, quality to this plate sound. Drum Example 9 (Video 18) and Drum Example 10 (Video 19) demonstrate the sounds of the membrane being destroyed and then reconfigured, and being struck and shaken during the process. In video 18, one can hear the instrument sloshing around, as particles are disconnected from one another, leaving parts of the membrane free-floating. Video 19 demonstrates what it sounds like to scrape this instrument once it has been partially destroyed.

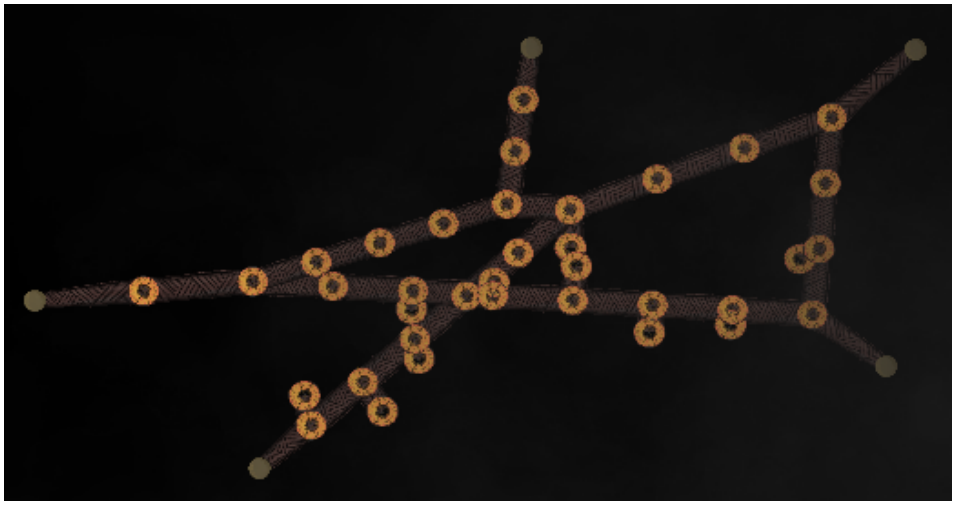

Figure 6.19: Non-realistic vibrating body modeled in Ruratae

6.3.3 Non-realistic vibrating body

The non-realistic vibrating body (shown in fig. 6.19) is composed of several strings bound together in a haphazard formation, with stray particles connected along the strings, which act as noise makers. This model is a perfect example of the possibilities of the system in the realm outside of realistic physical models. Thing Example 1 (Video 20) demonstrates a slow scrap around the object. One can hear the rapid changes of modal composition as the user scrapes along the body's outline. Thing Example 2 (Video 21) plucks the body from various positions. Each pluck produces buzzing noises which are produced from the free-ended particles coupled to the strings. These free-ended particles attempt to stabilize and dampen dependent solely on their relative position to the strings. Thing Example 3 (Video 22) reduces the model's muffling quality, producing a very noisy and resonant object. Thing Example 4 (Video 23) demonstrates how rigidness affects the object's sound quality. Increasing the rigidness makes the object even more noisy, since the free-ended particles have more spectral agency when their springs have non-zero resting length. Thing Example 5 (Video 24) continues examining how rigidness affects this non-realistic body. The model is tightened and loosened, at which point the listener can hear how the object's modes are transformed during this deformation. Thing Example 6 (Video 25) demonstrates an attempt

to scrape the object, now that it is fairly rigid. This produces rapid changes in the modal characteristics of the body, producing noisy, sloshing sounds which are reinforced by the free-ended particles. Thing Example 7 (Video 26) couples this non-realistic object with the previous harp object, connecting the object with one of the harp's string via a newly constructed spring. This example demonstrates how responsive the system can be when coupling the vibrations between two objects. Depending on how this spring transfers energy between the two systems, it will produce various results. This coupling behavior offers users the ability to create huge multi-instrument models that can be coupled at explicit locations, a concept made popular by digital waveguides.

6.4 Implications for Interactive Media

Since its inception into academia, computer music contributions often nod to its traditional role as tool provider to fixed media artists. Though certainly this particular tool has clear benefits for these sorts of artists, its real strength lies in non-linear, interactive media. This system has already demonstrated its strength as a real-time construction and performance interface. Though it can be beneficial to composers, sound designers and performers that work in either fixed or linear media paradigms, its real power lies in interactive, non-linear media, such as installation art and video games.

There has been a significant amount of research into using dynamic computational acoustics models in video game engines [7]. This has primarily been a task focused on wave propagation models and realistic, 3-D audio. Though this domain is certainly interesting, there has been very little use of simulation of mechanical vibrations. Historically, video games have implemented Newtonian simulations of particle and rigid bodies [14, pg. 2]. These simulations are designed to model relatively large bodies moving relatively slowly, focusing more on gravity and body collisions instead of oscillatory motion. The system presented in this text hopes to offer an extension to these types of physics engines by allowing video games to properly simulate the complex vibrational modes of objects in a game world.

These objects can vibrate and produce sonic energy relative to how they collide with one another in the game world. Additionally, when objects are destroyed, the engine can be used to simulate the sound of their destruction. Ideally, an artist would simply need to submit a mesh model with attached metadata information to the audio engine and it would simulate its interactions accordingly. Obviously, it is considerably more expensive to simulate sonic vibration than forces that can be integrated as the visual frame-rate. Therefore, care should be taken to only simulate vibrational modes of objects within a limited vicinity of the listening position.

Chapter 7

Conclusion

I have shown that it is possible to develop a system that provides users an interface to physically-informed virtual instruments that does not prerequisite expert knowledge of acoustics or DSP in order to use. The system is determinate, and given enough cues, users can develop deeper understandings of the system's behaviors through repeated interaction. There is room to improve on how users manage control paramters; developing a CAD-like editor would benefit users in the long-run to be able to describe and shape models more effectively. The visual interface depicting physical displacement effectively demonstrates vibrational behaviors, but developing more intuitive and elegant ways to describe parameters visually will benefit the system. Many of these improvements can arise by enabling users to develop their own rendering shaders; to enable advanced users to develop catered renderings of the system. In the future, this could be accomplished through a well-documented plugin system.

System control could be improved on as well. Currently, the demo software only supports mouse input; ideally this should be generalized through a universal input plugin system that would enable any and all sorts of inputs to be handled agnostically. This could be accomplished by providing an input plugin with a reference to the model, enabling the plugin to query the model for relevant information such as a the nearest particle to the input's virtual coordinates, as well as allowing the input plugin to modify the model parameters or inject impulses and force vectors into the model. When generalized, this structure would provide

almost any possible input device the ability to interface with Ruratae.

The system is fairly optimized but there are many improvements for real-time computation that can still be leveraged. It needs to be scalar, able to support a range of devices from mobiles to desktops to clusters, enabling a full range of possible use situations. SIMD Intrinsics are not fully optimized, lockless threading techniques could be exploited further and some tasks could be offloaded to the GPU. The largest hurdle to produce effectively-optimized code is still developing smarter ways to avoid cache misses, which impede the system worse than any other offender. Optimized and self-configuring memory structures are paramount in developing fast, efficient code.

Currently, the system is fairly stable, though creative users may find it too easy to create numerical explosions or infinitely self-impulsing systems. Using a combination of sophisticated integration methods and practical hotfix-style algorithms, the system can be stabilized more effectively. Stability gains from Beeman's method, coupled with running the system at as high a samplerate as possible will provide ample space for users to develop without worry of instability issues. Additionally, a simple energy-measuring thread can run, detecting any possible explosions as they occur and can enact a high drag force on the system until it becomes stable again.

Lastly, the system in its current state does not a straight-forward way for users to generate instruments based on fundamental frequency. A parallel thread that analyzes and measures the natural frequencies of linearized forms of the model will allow users to more accurately render models according to desired pitch or modal composition. One of the encountered pitfalls of this type of system is that users coming from a spectral-composition background feel that the system lacks the ability to generate based on desired pitch. This is a two-fold problem; part of the solution involves re-educating these users to understand how mechanical systems work. The other part of the solution involves developing smarter editor instructions to allow the system to respond to such pitch-based requests by analyzing and reconfiguring models automatically. The problem is particularly complex, and I predict a large portion of future development will be dedicated to tackling this

problem.

Throughout this document, I have demonstrated the historical underpinnings, the mathematical framework, the specifications and implementations and finally the realized practical application of this proposed physics-based audio engine. I have shown what necessitates the power and accessibility this system affords to its end-users. I have described the known problems and issues that arise when attempting to afford these abilities and I have offered several solutions and strategies that can be employed to tackle and reduce these upsets. In the near-future, it is my aim to fully realize the technology discussed, and to distribute to a wide general audience, in hopes that this contribution can impact people in many expressions of creativity and play. In summary, the system has accomplished many of its initial tasks but there is significant room for improvement in dealing with user interaction and usability.

Bibliography

[1] David G. Alciatore. *Introduction to Mechatronics and Measurement Systems*. McGraw Hill, 2007.

[2] Ralph Baierlein. *Newtonian Dynamics*. McGraw-Hill, New York, NY, 1983.

[3] David Beeman. Some multistep methods for use in molecular dynamics calculations. *Journal of Computational Physics*, 20(2):130,139, 1976.

[4] Stefan Bilbao. *Numerical Sound Synthesis*. Wiley, United Kingdom, 2009.

[5] Robert Bridson. *Fluid Simulation for Computer Graphics*. A.K. Peters, Ltd., Wellesley, MA, 2008.

[6] Claude Cadoz, Annie Luciani, and Jean Loup Florens. Cordis-anima: A modeling and simulation system for sound and image synthesis: The general formalism. *Computer Music Journal*, 17(1):19,29, 1993.

[7] Karen Collins. *Game Sound*. MIT Press, 2008.

[8] Neville H. Fletcher and Thomas D. Rossing. *The Physics of Musical Instruments*. Springer, New York, 2nd edition, 1998.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, 1995.

[10] Jason Gregory. *Game Engine Architecture*. CRC Press, Baco Raton, FL, 2009.

[11] Francisco Iovino, René Caussé, and Richard Dudas. Recent work around modalys and modal synthesis. 1997.

[12] Eric Lengyel. *Mathematics for 3D Game Programming and Computer Graphics*. Course Technology, China, 3rd edition, 2012.

[13] A. E. H. Love. *A Treatise on the Mathematical Theory of Elasticity*. Cambridge Press, 1906.

[14] Ian Millington. *Game Physics Engine Development*. CRC Press, Baco Raton, FL, 2nd edition, 2010.

[15] Branislav Nikolic. Computational methods of physics lecture notes.

[16] Julius O. Smith. Physical modeling using digital waveguides. *Computer Music Journal*, 16(4):74,91, 1992.

[17] Julius O. Smith. Physical modeling synthesis update. *Computer Music Journal*, 20(2):44,56, 1996.

[18] Eric W. Weisstein. Damped simple harmonic motion.