Pianist's Hands: Synthesis of Musical Gestures

Dissertation

zur

Erlangung der naturwissenschaftlichen Doktorwürde
(Dr. sc. nat.)

vorgelegt der

Mathematisch-naturwissenschaftlichen Fakultät

der

Universität Zürich

von

Stefan Müller

von

Rickenbach LU

Begutachtet von

Prof. Dr. Peter Stucki
PD Dr. Guerino Mazzola

Zürich 2004

Die vorliegende Arbeit wurde von der Mathematisch-naturwissenschaftlichen Fakultät der Universität Zürich auf Antrag von Prof. Dr. Peter Stucki und Prof. Dr. Klaus Dittrich als Dissertation angenommen.

Abstract

The process of music performance has been the same for many centuries: a work was perceived by the listening audience at the same time it was performed by one or a group of performers. The performance was not only characterised by its audible result, but also by the environment and the physical presence of the performing artists and the audience. Further, a performance was always unique in the sense that it could not be repeated in exactly the same way.

The evolution of music recording technology imposed a major change to this situation and to music listening practise in general: a recorded performance suddenly became available to a dramatically increased number of listeners, and one could listen to the same performance as many times as desired. However, in a recorded music performance, the environmental characteristics and the presence of the performing artists and the audience are lost. This particularly includes the loss of musical gestures, which are an integral part of a music performance. The availability of electronic music instruments even enforces this loss of musical gestures because the previously strict connection between performer, instrument, and listener is blurred.

This thesis deals with the problem of the construction of musical gestures from a given music performance. A mathematical model where musical gestures are represented as high-dimensional parametric *gesture curves* is introduced. By providing a number of mathematical operations, the model provides mechanisms for the manipulation of those curves, and for the construction of complex gesture curves out of simple ones. The model is embedded into the existing performance model of mathematical music theory where a musical performance is defined as a transformation from a symbolic score space to a physical performance space.

While gestures in the symbolic domain represent abstract movements, gesture curves in the physical domain reflect "real" movements of a virtual performer, which can be rendered to a computer screen. For the correctness of the movements one has to take into account a number of constraints imposed by the performer's body, the instrument's geometry, and the laws of physics. In order to satisfy these constraints a shaping mechanism based on Sturm's theorem for cubic splines is presented.

A realised software module called the *PerformanceRubette* provides a framework for the construction and manipulation of gesture curves for piano performance. It takes a music performance and given constraints based on a virtual hand model as input. The resulting output consists of sampled physical gesture curves describing the movements of the virtual performer's finger tips. The software module has been used to create animated sequences of a virtual hand performing on a keyboard, for the animation of abstract objects in audio-visual performance applications, and for gesture-based sound synthesis.

Keywords: Gestural Performance, Performance Interfaces, Performance Theory, Computer Animation.

Kurzfassung

Während Jahrhunderten stand im Kern einer Musikaufführung, dass diese von den Zuhörern zur gleichen Zeit wahrgenommen, wie sie von den Interpreten wiedergegeben wurde. Eine Aufführung war neben ihrer akustischen Realität auch durch ihre Umgebung und durch die Präsenz der Interpreten und der Zuhörer charakterisiert. Zudem galt jede Aufführung als Unikat und konnte nicht exakt in der selben Form wiederholt werden.

Die Möglichkeit, eine Aufführung akustisch aufzunehmen, auf replizierbaren Medien zu speichern und diese an beinahe beliebigen Orten wiedergeben zu können, eröffnet das Tor zu einem vollkommen neuen Umgang mit Musik und deren Aufführung. Erstens erreicht eine bestimmte Aufführung ein wesentlich höheres Zielpublikum als zuvor, und zweitens kann diese beliebig oft in der selben Form wiedergegeben werden. Da sich der Aufnahmeprozess jedoch nur auf die akustische Realität bezieht, gehen die Eigenschaften der Umgebung und die Handlungen der Interpreten und des Publikums sowie deren Wechselwirkungen bei der Aufnahme verloren. Dieser Verlust beinhaltet auch die musikalische Gestik, welche ein integraler Bestandteil einer Interpretation ist. Mit der Verfügbarkeit von elektronischen, computer-gestützten Musikinstrumenten wird der Verlust der Gestik weiter verstärkt, da die zuvor feste Verbindung zwischen Zuhörer, Interpret und Instrument zu einem grossen Teil aufgelöst ist.

Die vorliegende Arbeit befasst sich mit dem Problem der Konstruktion musikalischer Gesten aus der Interpretation einer gegebenen Partitur. Im Zentrum steht dabei ein mathematisches Modell, in welchem Gesten als hoch-dimensionale parametrische *Gestenkurven* repräsentiert werden. Eine Anzahl von mathematischen Operationen ermöglicht die Manipulation dieser Kurven und die Konstruktion von komplexeren Gesten aus einfachen Basisgesten. Das Modell ist eine Erweiterung des bestehenden Modells der mathematischen Interpretationstheorie, welche eine Interpretation als Transformation von einem symbolischen Partiturraum in einen physikalischen Aufführungsraum definiert.

Während Gesten in der symbolischen Realität abstrakte Bewegungen repräsentieren, stellen Gestenkurven in der physikalischen Realität reale Bewegungen eines virtuellen Interpreten dar, welcher mit Hilfe der Computergrafik am Bildschirm dargestellt werden kann. Die Korrektheit der Bewegungen wird durch eine Anzahl Bedingungen gewährleistet. Die Bedingungen sind durch die Anatomie des Interpreten, der Geometrie des Instrumentes und durch die physikalischen Gesetze gegeben. Der Sturmsche Satz über die Nullstellen von Polynomen hilft dabei, die Kurven so zu verformen, dass sie den gegebenen Bedingungen genügen.

Das Softwaremodul *PerformanceRubette* ermöglicht die Konstruktion und die Manipulation von Gestenkurven für Klavierinterpretationen. Ausgehend von einer interpretierten Partitur und den Rahmenbedingungen eines computergestützten Handmodells werden dabei die Bewegungskurven der Fingerspitzen des Interpreten berechnet. Diese können von einer Animationssoftware importiert und zur Animation des Handmodells verwendet werden. Zudem können die Bewegungskurven auch zur Animation von abstrakten visuellen Objekten in einer virtuellen Umgebung oder zur gesten-basierten Steuerung von Synthesizern verwendet werden.

Acknowledgments

This work was carried out at the MultiMedia Laboratory (MML) of the University of Zürich's Computer Science Department. I thank Prof. Dr. Peter Stucki who has set a milestone with the foundation of the MML and who constantly supported my work. I also thank PD Dr. habil. Guerino Mazzola. His enormous efforts in systematically applying mathematics to music have "modernised" the field of modern music research and his way of thinking has been a constant source of inspiration to me. I thank all my colleagues and friends at the MML for providing a stimulating and lively working environment: Dr. Stefan Göller, Dr. Mauro Iacobacci, Gérard Milmeister, Dr. Hansrudi Noser, Dr. Christian Stern, Dr. Marcia Ponce de León, Dr. Jody Weissmann, and PD Dr. Christoph Zollikofer.

My thanks go to Prof. Dr. Luc Van Gool and his group at the Computer Vision Laboratory at ETHZ for the permission to use their excellent hand model.

Particular thanks are due to my friends at Corebounce, Pascal Müller, Dr. Simon Schubiger, Matthias Specht, and Christian Widmer: the co-operative, endless programming sessions, and the numerous unforgettable live performances have definitely lead to new insights where live multimedia art stands at present, and where it may lead to in future.

Finally, I thank my parents who always supported my work and who have always been there when "little big city" was getting too big and I needed a few quiet days in the countryside.

Contents

Chapter 1

Introduction

For many centuries, the process of music performance has been the same: the work was performed by one or a group of performers, and perceived at the same time by the listening audience. A performance was always unique in the sense that a live performance could not be reproduced in exactly the same manner as a previous performance. Moreover, a performance was not only characterised by its audible result: the environment, e.g., a concert hall, and physical presence of the performing artists and the listening audience accounted for the originality of every performance.

The availability of music recording devices – the phonograph was invented by Thomas Edison in 1877 – had two fundamental effects on the established nature of music performance: First, the possibility to record, store, and play back a performance dramatically increased the number of listeners, one could listen to the performance without actually being there. Consequently, the number of listeners became much larger than the number of people who are doing music, and today music is mostly produced to be listened to. Second, a music performance could be played as many times as desired, and important parts could be selected and repeated, or be compared to other performances. In addition, there was finally a way to archive performed music.

However, the evolution of recording technology opened a gap that has not been closed to these days: Only the acoustic aspect of a performance is recorded, either in analogue form, as on vinyl discs, or digitally sampled and encoded, as on compact disks. Other integral parts of the performance are lost.

One of those lost aspects, and maybe the most important one, lies in the nature of recording and reproduction itself: the "feeling" of actually anticipating a live performance *now*, the fact that other people are performing a piece, and that other listeners are present, can not be recorded with today's technical capabilities. For instance, this also includes the human interaction between conductor and performers, performers among themselves, and performers and listeners.

Another aspect is concerned with the environment where a performance takes place. For instance, a concert recorded in an auditory will have a completely different effect when listened to at home. While there has been a lot of success in reconstructing a part of

the acoustic room properties when playing back a performance, the visual impression of the environment is lost.

Finally, the gestural part of the performance is lost. In a live performance, the listener has a clear relation of the performer's body movements, the physical interaction with the instrument, and the acoustic signals being produced.

At this point we may argue that the recording of above aspects can be covered by the ability to record the performance on video (e.g., on a DVD) as well. This is in part true for the second and the third aspect, recording a concert on video and play it back on TV is in fact a wide-spread practice. However, another development in audio technology requires more than just video-playback of a performance: the capability to produce sound synthetically, e.g., via synthesisers and computers, blurs the previously strict connection between environment, performer, instrument, and listener (Iazzetta, 2000). In the most drastic case, where the computer performs a piece, every aspect of human interaction is simply not existing, or took place in the effort to program the computer.

Thus, if we strive for making computer-generated, or computer-aided music performance more realistic we will have to deal with the reconstruction of the mentioned lost aspects. Ideally, this would include the reconstruction of all sensible realities, but mainly we would have to deal with visual and audible components of a performance.

1.1 Synthesis of Musical Gestures

This thesis discusses the construction of musical gestures. It deals with the problem of how we can synthesise gestures from given musical material in such a way that the gestures can be used for animation of a virtual performer playing an instrument. The way to a solution is very interdisciplinary, covering issues from musicology, music performance theory, playing technique, sound synthesis, and computer animation. First of all, a model for musical gestures, or gestures in general, will be required. As we shall see, only a few attempts have been made to formalise gestures in such a way that the model becomes available for computing. Ideally, the model should neither be dependent on an instrument, nor on the performer playing it. In addition, the model should be valid for different types of gestures, e.g., for movements related to playing the instrument, as well as for (musically) completely independent gestures. The model should also allow the construction of complex gestures from simple ones in order to build grammars of gestures at different physical and mental levels. Once the model has been defined, we will need mechanisms (i.e., algorithms) for the transformation of the given musical material to gestures, and vice versa. Additionally, the synthesised gestures should be available to be used by computer animation tools, e.g., for controlling a performer in a virtual environment.

In our approach, the model of musical gestures can be seen as an extension of existing research in mathematical music theory (Mazzola, 2002c), and performance theory in particular, where a performance can be seen as a transformation from a symbolic space (e.g, the score) to a physical space (e.g., the acoustic result of the performance). This situation is reflected in our model, there are symbolic gestures, representing abstract movements of a performer in the symbolic domain, and physical gestures representing

movements of a performer playing on an instrument in the physical domain. In our model, gestures are based on parameterised curves of arbitrary dimension. The spaces containing these curves constitute the individual properties of the particular space being required, either in the symbolic, or in the physical domain. Further, we will present operations on gesture curves, for example combination, concatenation, scaling, etc., thereby providing a mechanism for building complex gestures from gesture primitives.

From the viewpoint of computer animation, the automatic or semi-automatic generation of animated sequences has become immensely important over the last few years. At the time of writing this thesis, the third part of the movie "The Lord of the Rings" is running at the theatres. Sequences with thousands of computer-animated soldiers and monsters are impossible to be animated manually, and the there is a large variety of software tools for automated locomotion generation. Our method contributes to this development in the sense that computer-aided synthesis of musical gestures for human hand models can be applied in the same way to other types of gestures, for instance for general hand movements, for hand writing, or for hand sings, as they appear in sign languages.

While our model itself is not limited to a specific instrument, its realisation, in terms of implementation as a computer program, is focused on finger movements in piano performance. The results, motion curves of the finger tips for a given performance, can be used as input for a virtual hand model. This procedure can be seen as a computational pre-stage to further computer animation steps, such as inverse kinematics handling, or skinning.

One obvious application of synthesised gestures is, as mentioned above, the animation of virtual performers to make music performance more realistic. In addition, the availability to construct gestures from musical scores and use them to animate virtual performers is very helpful for the creation of educational music tools. Another useful application are gesture-controlled synthetic instruments, where our model can be seen as the formal glue between gestural control interfaces and synthesisers. In addition, the model can be used for composition, and for general visual accompaniment to music performance.

1.2 Thesis Overview

After this introduction, chapter 2 will present the background that has been the foundation of our model of musical gestures and its realisation. This includes musicological topics, such as definitions of musical gestures and music notation, an introduction to music performance theory and an overview on so-called Denotators and Forms, which account for the mathematical basis of our model. Further a short outline on different sound synthesis methods is given. The section on human hand models gives an overview on the anatomy of the human hand, and presents existing work of human hand models in computer graphics and animation. The concluding two sections deal with the platforms where our model has been realised: the first is a music research platform where such new models can be implemented in a convenient way. The second is a multimedia framework where synchronised audio and video applications can be modelled in real-time.

Chapter 3 presents our model of musical gestures. This includes the definition of

gestures as parameterised curves. Different elementary operations on the curves are presented. Then, we will show how gesture curves can be constructed from a (symbolic) musical score, and how they can be shaped according to constraints given by a hand model in the physical domain.

The realisation of our model will be presented in chapter 4. This chapter will present the architecture of Distributed Rubato, and several software components that were helpful for the implementation of the gesture model. It will then point out the detail design of the PerformanceRubette, a software component which is part of Distributed Rubato. The PerformanceRubette is a tool for music performance research, it serves as the basic implementation of the theory given in (Mazzola, 2002c) with the addition of our model of musical gestures. To conclude, we will present the algorithms that have been implemented to construct and shape musical gesture curves.

In chapter 5, we shall present realised and possible applications of the presented theory and implementation. First, we shall show how gesture curves have been used to animate a human hand model with the help of computer animation software. Related to the animation of a human hand model based on gesture curves is the generation and animation of abstract objects in interactive live performance tools: here gesture curves provide a powerful mechanism to performing video-artists. Then, we shall point out how gesture-controlled sound synthesis can be realised. We will conclude with a section on gesture-based musical composition.

Finally, chapter 6, contains a recapitulation of the basic ideas presented in this thesis. It will highlight the solutions and problem areas and give an outlook on possible directions for future research and development.

Chapter 2

Background

This chapter provides the reader with the necessary background required for understanding the theory and the implementation presented in the subsequent chapters. It takes the multi-disciplinary character of this thesis into account and starts out with musicological background on musical gestures, music notation, and the theory of music performance. Then an introduction to Denotators and Forms is given, these concepts provide the mathematical foundation for our theory of musical gestures. Section 2.5 will give an overview on current sound synthesis methods, with special focus on physical modelling. It is assumed that the reader has basic signal-processing knowledge. The chapter then continues with an introduction to human hand models, and in particular presents existing work on hand models in the domain of computer graphics and animation. The concluding two sections present two software frameworks that were used to realise (i.e., implement) the theory on musical gestures.

2.1 Musical Gestures

In almost every situation of human behaviour and communication gestures are present. Often one is not aware of the expressive power of gestures since many of them are not actively realised by our perception. Generally, any body movement can be seen as a gesture, it may be a long-trained action to perform a specific goal on the one hand, or an uncontrolled reflex movement on the other:

> **gesture**, *n.*
> **1. a.** Manner of carrying the body; bearing, carriage, deportment (more fully ***gesture of the body***); *rarely* in *pl. Obs.* (mergend in 3).
> **b.** Grace of manner. Also *pl. Obs.*
> **2. a.** Manner of placing the body; position, posture, attitude, *esp.* in acts of prayer or worship. Also, specified posture. *Obs.*
> **3. a.** In early use: The employment of bodily movements, attitudes, expression of countenance, etc., as a means of giving effect to oratory. *Obs.*

> **b.** Now in narrower sense, as a generalised use of 4: Movement of the body or limbs as an expression of feeling.
> **4. a.** A movement of the body or any part of it. Now only in restricted sense: A movement expressive of thought or feeling.
> **b.** *transf.* and *fig.*; *spec.* [after F. *geste*; cf. beau geste] a move or course of action undertaken as an expression of feeling or as a formality; esp. a demonstration of friendly feeling, usu. with the purpose of eliciting a favourable response from another.
> **5.** *attrib.*, as ***gesture language, -sign, -speech, -syntax***; **gesture theory**, a theory of the origin of language; hence ***gesture-theorist***.
> (*The Oxford English Dictionary*, Oxford University Press, OED Online).

In this section, we shall discuss different attempts to define and classify gestures with particular focus on musical gestures. Further, we will point out the shift that took place when recording of music became available, a shift that took place in the physical (acoustic) reality, and which has its counterpart in the symbolic (score) reality. We will conclude with observations on gesture grammars and gestural primitives, which can provide a formal framework for human motion in general.

2.1.1 Definitions and Classifications

In (Cadoz and Wanderley, 2000) different definitions of gestures from human – human, and human – computer interaction literature are discussed. It becomes clear that there is no single definition that covers all aspects related to gestures, either the definitions are very general and not specific enough for certain fields, or they are too specialised and only valid in their own context but not any longer in an other domain. Common to many definitions is the notion that gestures are body movements that contain information and produce meaning; for instance:

> *Gesture:* for the purpose of this paper it is sufficient to understand "gestures" as body movements which convey information that is in some way meaningful to a recipient (Wachsmuth, 1999, 277).

The production of sounds intimately involves human motion. For the classification of musical gestures Ramstein (1991) has proposed an analysis of instrumental gestures consisting of three approaches: A phenomenological approach, a functional approach, and an intrinsic approach. The *phenomenological* approach, a descriptive analysis, is based on a number of criteria: *cinematic* (analysis of the movement speed), *spatial* (size of the space where the gesture takes place), and *frequential* (movement decomposition regarding its periodic content). As the next step, the *functional analysis* refers to the possible functions a gesture may perform in a specific situation. The three distinct types of gestures in this approach are the *effective gestures* (mandatory movements to mechanically produce sound), the *accompanist gestures* (body movements associated to effective gestures), and the *figurative gestures* (gestures unrelated to the purpose of sound production). Finally, the *intrinsic* approach is based on the performer's viewpoint of producing gestures, i.e.

the hand as a body part with fine motor skills and a number of nervous receptors. Then the diversity of trajectories and gestural behaviours within a system (fingers, hands, feet) is analysed.

From the viewpoint of our work, it is important to point out that we are looking for a formal definition of musical gestures that will fit in a mathematical framework. We will give our own definition in section 3.1, where gestures are defined by means of parameterised curves residing in an arbitrary space. Gestures in our framework can therefore also represent movements not only in physical space, but also in mental spaces, for instance. Mental gestures (Zagonel, 1992) are closely related to composition and listening. A composer often starts with a mental image of sound gesture to compose a vocal or instrumental gesture. Listeners, on the other hand, often mentally recreate the performer's physical gestures while listening to music.

2.1.2 The Loss of Gestures in Recorded Music

An important observation was made in (Iazzetta, 2000) that the appearance of electronic and recorded sound induced a shift in the habits of listening to music:

> For many centuries, people learned to listen to sound that had a strict relation to the bodies that produced them. Suddenly, all this listening experience accumulated during the long process of musical evolution was transformed by the appearance of electronic and recorded sounds. When one listens to artificially generated sounds he or she cannot be aware of the same type of concrete and mechanic relations provided by traditional acoustic instruments since these artificial sounds are generated by processes that are invisible to our perception. These new sounds are extremely rich, but at the same time they are ambiguous for they do not maintain any definite connection with bodies or gestures (Iazzetta, 2000, 259).

Thus, we may say that *gestures are lost in recorded music.* While this happened in the physical (acoustic) reality of music performance, we will find a parallel development in the symbolic (score) reality in section 2.2: music notation originated from writing down gestures, but its evolution lead to ever higher abstraction levels. In today's Western music notation, only the abstract score symbols remain and gestures are not explicitly present any longer.

2.1.3 Gestural Primitives and Gesture Grammars

If we want to apply computational methods to gestures and gestural performance, we first need a formalisation of gestures in terms of gesture grammars and gestural primitives. With an eye on the enormous complexity of human motion, it becomes clear that one has to start out with elements (i.e., the primitives) characterising simple motions, and then construct more complex gestures built of those elements, eventually resulting in a grammar of gestures.

While there has been extensive research in other gesture-related fields, for instance in *American Sign Language* (ASL) (Liddell, 2003), or in *Human Computer Interaction* (HCI) (Weissmann, 2003), much less work has been done in the area of musical gestures.

One of the exceptions is (Choi, 2000), where attempts are made to bring a formalisation of human motion in terms of performance gesture into the computable domain. Choi introduces and classifies a number of gestural primitives, which, in his context, are fundamental human movements that relate the human subject to dynamic responses in an environment. A gestural primitive consists of four elements: a n-dimensional phase space P of an input sensor; an initial motion Δ, which is a vector in P; a function $\lambda(t)$ describing change in Δ over time, identified as "observable" change of movement; and a physical model M that maps Δ and $\lambda(t)$ from phase space P to a performer's movement space. The physical model M consists of three classes of mapping between phase space P and movement space: rotation (change in orientation), gradient (linear change), and period (repetition). These changes account for the mechanical constraints of the input sensor, and for the physical disposition of the performer to the sensor. This formal framework is then realised in a software environment to facilitate multi-modal performances with interactive simulations.

As we shall see in section 3.2, we will define operations on gestures, that for instance allow manipulation and composition of basic gestures, in order to build more sophisticated ones.

2.2 Music Notation

This section gives an overview of the evolution of western music notation. Almost every culture with its own script has developed its own script for music notation. The notation signs are often closely related to the signs of the language script. While many types of early music were passed from generation to generation without being notated, the availability of music notation allowed for precision and consistency.

The Greek and Roman systems were non-graphical notations that used letters of their alphabets to symbolise notes. The occidental music theory is still in context with the Greek music theory, as it has been passed down by Boethius (c. 480 – c. 525 A.D.), a roman philosopher, poet and politician. Today's use of the letters A to G for naming a note of a certain pitch class, is sometimes called *Boethian notation* and has been adopted from Greek and Roman practise.

First examples of Christian notation are found in theoretical essays, such as the *Dasia notation* in *Musica enchiriadis*, or the letter notation of Hucbald of St. Amand. The oldest *Neumes* are found in the 10th century. Neumes (Parrish, 1957) are signs denoting a single pitch (*virga*, *punctum*), two pitch classes (*pes*, *clivis*), and three pitch classes, and have been used to support the consistency of handing down the Gregorian chorals. The word "neuma" is the Greek word for "hint", but is also used in the sense of "pneuma" (breath) and "nota" (clause). Prior to neumes, *cheironomy*, which means the use of hand gestures to indicate melodic shape, was used in different cultural contexts, such as in Ancient Egypt, in the Jewish tradition, in the Byzantine tradition, and in the occidental

choral.

The use of *bar lines* has been introduced for the first time by Guido of Arezzo in the 11th century, and this so-called *diastematic notation* made it possible to record the pitch of a note. In the 12th century, the neumes evolved into the *square notation*. The duration of a note was constituted by the pattern of subsequent two or three note ligatures, but it was not possible to denote the exact duration of a single note. Only with the introduction of the *mensural notation* by Franco of Köln in the 13th century it became possible to abstractly define the exact duration of a note. Still, the concept of bars was not introduced before the baroque age.

With today's western music notation we have a highly abstract method of writing down musical content, and, as just has been shown, the degree of abstraction has been ever increasing with the evolution of music notation over the centuries. This observation will lead us to the thesis in section 3.1.1 that symbols in a musical score can be seen as *frozen gestures*.

2.3 Theory of Music Performance

In (Mazzola and Zahorka, 1994b), a general theory of performance transformations $\wp$ from a symbolic score space S to a corresponding physical space P was given. The transformation $\wp$ can be represented by performance vector fields that generalise the well-known tempo curves of a performance. This theory has been implemented as a module in the music research software Rubato (Mazzola and Zahorka, 1994a) and has been successfully applied to the performance of classical compositions. Stange-Elbe (1999) has performed Contrapunctus III of J. S. Bachs *Die Kunst der Fuge* with Rubato, and the results have been presented at the Diderot Conference on Mathematics and Music at the *Institut de Recherche et Coordination Acoustique/Musique* (IRCAM) (Mazzola, 2002a). For a full account on performance theory, refer to (Mazzola, 2002c), parts VIII and IX.

2.3.1 Conventions and Definitions

Let us first introduce some conventions used throughout this thesis. First of all, it is crucial to understand that the theory deals with symbolic musical events (e.g., data retrieved from a score file) rather than acoustic signals. Thus, a score (or composition) C is a set of vectors (also referred to as points, notes, or events) in an n-dimensional vector space. The performance $C_{Performed}$ is the corresponding set of the performance events, i.e., the transformed score events.

The dimension of the vector spaces is arbitrary, because the presented theory is generic. Nevertheless, it is sufficient for the reader to assume that the vector spaces are rather simple and, for example, built of basic instrument parameters, such as onset time, pitch, loudness, and duration. In the symbolic score space S, we will typically use E for onset time (German "Einsatzzeit"), H for pitch ("Tonhöhe"), L for loudness ("Lautstärke"), and D for duration ("Dauer"). The corresponding lowercase symbols will be used for the performance space P.

When referring to a specific point in S, the symbol X will typically be used; the performance vector field at X will be named $Ts(X)$.

2.3.2 An Infinitesimal View on Expression

The precise description of a musical performance poses major difficulties. On the one hand, expression is a multi-layered semiotic phenomenon. That is, expression extends from surface parameters to more hidden structures that reveal the scores analytical depth structure, for example. We do not deal with this complex problem here, because on the other hand, the surface expressiveness is in-decomposable in general, i.e., it is typically not possible to separate expressive shaping of onsets (agogics) from duration (articulation), loudness (dynamics), or pitch (intonation). For example, the Chopin rubato makes it impossible to recover a single tempo curve, because the agogical structure depends on pitch when chords are slightly arpeggiated, and the right-hand melody onsets are locally deformed against the left hand accompaniment. One therefore needs a language that copes with this intrinsic intertwining of parameters. In traditional musicology, performance theory has never developed an adequate conceptualisation beyond fuzzy common language descriptions, although (Adorno, 1963) promoted an infinitesimal view of performance, termed micro-logic and based upon the insight that performance deals with the infinite interpolation of shaping parameters. This deficiency is typically reflected in the feuilletonistic music criticism and has to date prevented a truly scientific musicological performance theory. More specifically, inverse performance theory is far beyond musicological concepts, because the reconstruction of system parameters of a given performance would require a precise definition of the performance data and the system set-up. Because not even a performance theory based upon score analysis is available, such a system description remains out of reach of traditional musicology. However, in computer-aided performance research, system descriptions, which would enable inverse performance theory, have been proposed (Sundberg, 1991; Todd, 1992).

We should remark on the concept of expression, because its meaning is ambiguous. If we attempt to analyse expression, this regards not the psychological perception of a performance by humans. This aspect is a legitimate one, but it touches a category that relates the performed music to human categorisations in terms of emotional response. Such a perspective is dealt with, for example, by Honing (1992) and Langner et al. (2000). In contrast, we regard expression as a rhetorically shaped transfer of structural score contents by means of the deformation mapping of symbolic data into a physical parameter space. The psychological implications are not the subject of this perspective; it is a purely mathematical description of this mapping, not of the emotional correlates.

2.3.3 Performance Transformations

Before getting into general performance transformations, let us have a look at the symbolic, musical time E, which is associated with the physical time e. Let us suppose that the performance transformation $\wp$

$$E \to e \tag{2.1}$$

is an invertible C^1 (continuously differentiable) function $e = \wp(E)$. Then the *tempo curve* associated with $\wp$ can be defined as

$$T = \left(\frac{d\wp}{dE} \right)^{-1} \text{ [Beats/sec]}. \tag{2.2}$$

If we know the physical time of the start time $e_0 = \wp(E_0)$, then the transformation $\wp$ is the integral

$$\wp(E) = e_0 + \int_{E_0}^{E} 1/T. \tag{2.3}$$

Above concept of having a transformation $\wp$ for tempo can be generalised for an arbitrary number of musical parameters. Suppose that we are given the *composition* C of sound events in a score space S (for example $\mathbb{R}\{E, H, L, D\}$). Each element $X \in C$ is transformed into a physical sound event $x = \wp(X)$. The C^1 transformation $\wp$ can be described by performance vector fields which generalise the situation studied above for tempo. Consider the diagonal constant vector field Δ on $C_{Performed}$:

$$\Delta(x) = \Delta = (1, \ldots, 1) \tag{2.4}$$

for all $x \in C_{Performed}$. Then, the performance field $Ts(X)$ is the inverse image of the Δ field, as denoted in figure 2.1:

$$Ts(X) = J(\wp)^{-1}(X)(\Delta), \tag{2.5}$$

where $J(\wp)$ is the Jacobian matrix at X

$$J(\wp) = \left(\frac{\partial x_i}{\partial X_j} \right) \Bigg|_{X_j = E,H,L,D}^{x_i = e,h,l,d} (X). \tag{2.6}$$

2.3.4 Inverse Performance

The theory of *inverse performance* has been addressed in (Mazzola, 1995). It deals with reconstruction of performance characteristics from a given score and its performance(s). In (Müller and Mazzola, 2003b) a first step to inverse performance calculation was given: the calculation of performance fields from a given performance. The theory has been realised in a software module called the *EspressoRubette*. Basically, the problem can be split into two parts: first, the problem of matching symbolic and performance events, which has been extensively discussed in literature (Dannenberg, 1984; Vercoe, 1984; Puckette and Lippe, 1992; Heijink et al., 2000). Second, the calculation of the performance fields after matching has been performed is based on the calculation of local basis vectors, which are then used to calculate the Jacobian matrix for each score-performance event pair. The performance field vectors are then used for the calculation of an interpolated performance vector field at arbitrary resolution. The resulting performance field can then be used for vector field visualisation or for the comparison to other performance fields that have been calculated using a different performance, which can be seen as an effective method for performance calculation.

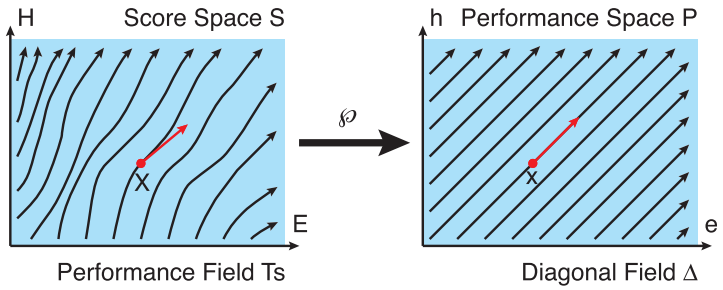

Figure 2.1: The performance transformation $\wp$ from a score space S to a performance space P may be described by the performance vector field Ts, and the diagonal field Δ.

2.4 Denotators and Forms

This section gives an overview of the universal data model of *Denotators* and *Forms*, which will be used to define our view on gestures in chapter 3. The model can be seen as a bridging interface between modern mathematics, programming theory, and systematic concept construction: first, Denotators and Forms are algebraic structures, the Denotator system shares properties of a mathematical *category* (for an introduction to category theory refer to (Asperti and Longo, 1991)). Second, the concepts of Denotators and Forms share the idea of object-oriented programming, such as abstraction and data-hiding. Third, Denotators and Forms are very explicit elementary signs in the semiotic sense, and allow for the construction of concept spaces.

For a full account and a further formal treatment of Denotators refer to (Mazzola, 2002c, chapter 6), for a semiotic approach refer to (Mazzola, 1999).

2.4.1 Forms

A Form is a named hierarchical or circular space, and a Denotator is a substance point residing in that space. Thus, a Denotator always comes with its corresponding Form. The general definition of a form is written as:

$$FormName \longrightarrow FormType(Coordinator). \tag{2.7}$$

Since we deal with named spaces, every form must have its unique name ($FormName$). To allow the definition of arbitrary hierarchical and circular spaces, different compound types ($FormType$) exist. The compound types define the structure of the Form's co-ordinator.

Limit. Limits include the definition of Cartesian products of the forms defined in the co-ordinator list. The co-ordinator is an ordered list, or an "n-tuple" of elements, of the n factor Forms.

Colimit. Colimits are disjoint unions of several object collections. In other words, the substance of a Colimit Denotator is a single Denotator selected from one of the given co-ordinators. The co-ordinator is an ordered list of n Forms of the collection.

Powerset. Powersets are sets of objects of a certain kind, which is given by its co-ordinator Form. Thus, they are not ordered.

Synonymy. With Synonymy we can rename a certain form given by the co-ordinator Form.

At the leaves of a form hierarchy, we have forms of type **Simple**, which allow the definition of basic types given in the form co-ordinator. The most commonly used basic types are STRING (the set of character strings), BOOLEAN (the set $BIT = \{0, 1\}$), INTEGER (the set $\mathbb{Z} = \{0, \pm 1, \pm 2, \pm 3, \ldots\}$ of integers, and FLOAT (the set $\mathbb{R}$ of decimal numbers).

Let us illustrate how a data-structure for a simplistic musical score can be defined using above concepts. First, a score is a set of notes:

$$Score \longrightarrow \textbf{Power}(Note). \tag{2.8}$$

Then, a note consists of four parameters, onset time E, pitch H, loudness L, and duration D:

$$Note \longrightarrow \textbf{Limit}(E, H, L, D).$$

Finally, we can define the four simple types:

$$E \longrightarrow \textbf{Simple}(FLOAT)$$
$$H \longrightarrow \textbf{Simple}(INTEGER)$$
$$L \longrightarrow \textbf{Simple}(STRING)$$
$$D \longrightarrow \textbf{Simple}(FLOAT).$$

Figure 2.2 illustrates this example Form by giving a graphical representation.

2.4.2 Denotators

Once we have a given form (referenced to by its name) we can define a denotator in the following way:

$$DenotatorName : Address \rightsquigarrow FormName(Coordinates). \tag{2.9}$$

Again, the $DenotatorName$ is a unique identifier as a reference to the Denotator. However, the Denotator names are of secondary importance only. Denotators can be anonymous, they can be referenced (and thus distinguished) to by their co-ordinates only. This is necessary when dealing with large numbers of Denotators of the same Form, such as the notes in a score.

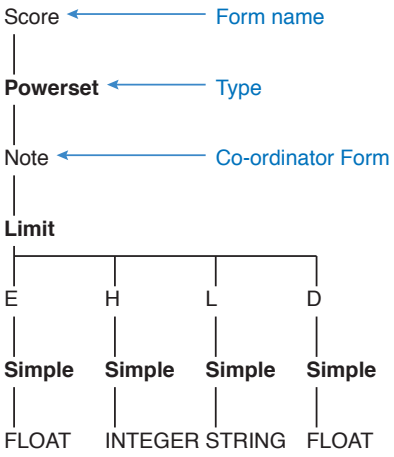

Figure 2.2: The simplistic $Score$ Form in graphical representation.

The address can be any space in the category used in the specific concept architecture. Often this is an algebraic module M. In our context, we mostly deal with topological spaces, such as the unit interval $I = [0, 1] \in \mathbb{R}$ or the zero address $0 = \{0\} \in \mathbb{R}$. The address concept is an important generalisation in the sense that variable addresses enrich the expressiveness of Denotators: the address carries additional information on how the Denotator has to be interpreted. We can see a Denotator as a "point" in a space (its Form) that is not a fixed one, but may vary as a function of an entire parameter system. Most commonly, we are dealing with zero-addressed Denotators, we then just write

$$DenotatorName \rightsquigarrow FormName(Coordinates) \tag{2.10}$$

thereby omitting the zero address. As we shall see in chapter 3, I-addressed Denotators, so that the "point" parameter is a real number $0 \leq \lambda \leq 1$, will be used to define Denotators of parametric curves.

Depending on the type of the Form F of a given Denotator f the co-ordinates have a specific definition, which is also reflected by the notation:

Limit. If the co-ordinator of F consists of a list of Forms $F_1, \ldots, F_n$, we define the co-ordinates of f to be any sequence $f_1, \ldots, f_n$ of Denotators with forms $F_1, \ldots, F_n$:

$$f = fName \rightsquigarrow F(f_1, \ldots, f_n)$$

with

$$\begin{aligned} f_1 &= f_1 Name \rightsquigarrow F_1(f_1 Coordinates) \\ &\ldots \\ f_n &= f_n Name \rightsquigarrow F_n(f_n Coordinates). \end{aligned}$$

Colimit. For the colimit, with a given co-ordinator $F_1, \ldots, F_n$, we define the co-ordinates of f as being any Denotator f_i of Form F_i:

$$f = fName \rightsquigarrow F(f_i)$$

with

$$f_i = f_i Name \rightsquigarrow F_i(f_i Coordinates)$$

for index $i = 1, \ldots, n$.

Powerset. For the powerset, with a given co-ordinator F_P, the co-ordinates of f are defined as a finite set $S = \{f_1, \ldots, f_n\}$ of Denotators, all having the same Form F_P:

$$f = fName \rightsquigarrow F(\{f_1, \ldots, f_n\})$$

with

$$f_k = f_k Name \rightsquigarrow F_P(f_k Coordinates)$$

for $k = 1, \ldots, n$. Usually, the curly set brackets are omitted in this notation, and we simply write:

$$f = fName \rightsquigarrow F(f_1, \ldots, f_n).$$

Synonymy. If the type is synonymy, and the Form F's co-ordinator is F_S, i.e., $F \longrightarrow$ **Syn**(F_S), we define the co-ordinates of f as being any Denotator f_S of Form F_S:

$$f = fName \rightsquigarrow F(f_S)$$

with

$$f_S = f_S Name \rightsquigarrow F_S(f_S Coordinates).$$

Simple. If the type of F is simple, the co-ordinates of f are directly defined by its value:

$$f = fName \rightsquigarrow F(fValue)$$

With above definitions, we can now explicitly write down an example score for the $Score$ Form defined in the example above:

$$myScore \rightsquigarrow Score(\{myNote1, myNote2\})$$
$$myNote1 \rightsquigarrow Note(1.0, 60, \textit{"ff"}, 1.0)$$
$$myNote2 \rightsquigarrow Note(2.0, 62, \textit{"fff"}, 2.5).$$

This example illustrates how values of the simple types (E, H, L, D) are implicitly integrated into the co-ordinates of the $Note$ Form. This is mainly done for practical reasons since it allows for a more compact notation.

2.5 Digital Sound Synthesis

In this section, an introduction to sound synthesis in general and background on different sound synthesis techniques is given. We shall particularly focus on techniques that apply for real musical instruments, and how gestural parameters can help to control the synthesised sound. We will further present the physical modelling software *Modalys*. Sound synthesis is a fairly broad field and an excellent introduction and reference can be found in (Roads, 1996), for an introduction to digital sound production refer to (Müller, 1998).

2.5.1 Synthesis Basics

Before the 1950s, sound synthesis was exclusively achieved in the *analogue domain*: analogue electronic signals were for instance generated trough oscillating vacuum and gas tubes and then modified using analogue filter circuits. The unique sounds of those devices has kept them alive to present, and they are still widely used among sound engineers and musicians. Later, the vacuum and gas tubes were replaced with analogue semiconductor-based circuitry. However, with the rapid developments in computer technology and in *Digital Signal Processing* (DSP) we can observe a considerable success of digital synthesis methods over the past decades. Nowadays, digital frequency modulation and wavetable synthesisers are integrated in virtually any personal computer that comes with a sound card. From the viewpoint of digital sound processing, a digital synthesiser can be seen as a *unit generator*, which takes a number of sample streams and control signals as input, and produces a number of sample streams as output. A *sample* is the basic unit of a discrete, digital signal. Some of those digital sound synthesis methods mimic the circuitry of their analogue counterpart, while others have been developed in the digital domain only and do not have an analogue pendant.

Before looking at different synthesis methods, it is very important to think about the goals to be achieved: do we want to synthesise existing instruments (either physical or electrical), or do we want to create a completely original sound, or do we even want to create ambient sounds that do not have instrument character? Sometimes it is desirable to have one synthesis method for a broad range of sounds and sound colours, in other cases there is a very specific application, for instance if we just wanted to synthesise the sounds of a wooden flute. Typically, different synthesis methods apply just for certain classes of problems, and there is no "one method fits it all" solution.

Another crucial issue is how the synthesis process can be controlled: most synthesis methods offer a large parameter space, but only a small subset of that space leads to acceptable results. A possibility is to provide a number of parameter pre-sets, which are designed for specific sounds – a technique used in many commercial synthesisers. Most synthesisers can be controlled trough the *Musical Instrument Digital Interface* (MIDI), and native synthesiser parameters can be set using vendor-specific MIDI messages. At this point, and with respect to this thesis, we raise the question, how gestural concepts and parameters can be incorporated into a certain synthesis method. As we shall point out in section 3.1, MIDI can be seen as a very simple gestural interface: the concepts of "Note on", "Note off" and in particular "Velocity" model a virtual performer playing on a

virtual keyboard. While this model is a good starting point, the interface for more complex gestural models needs to be improved. For instance, in (Laczko, 2003) a gestural interface for piano mechanics coupled to physically modelled strings was realised. We can expect that incorporating gestural parameters into synthesis methods will lead to more realistic sound synthesis. As an example we can take a gesture consisting of a whole phrase, i.e., a number of notes: synthesis of phrases is one of the big challenges in this field, and will become more important in the years to come.

2.5.2 Synthesis Methods

Today, there exists a large number of different approaches to sound synthesis. In addition, many commercial synthesisers incorporate more than one method, or come with hybrid approaches; and in most cases additional digital effects, such as reverberation, chorus, or echoes, can be appended to the synthesis output. We will give short descriptions of the most prominent synthesis methods, for details, the reader may refer to dedicated literature given in the general references.

Wavetable Synthesis

The basic idea of wavetable synthesis (Bristow-Johnson, 1996) is to use existing sound material (the wavetables) as a starting point, and then to create convincing simulations of acoustical instruments. In its early forms, fixed-size wavetables were pitch-shifted and then just looped, or sometimes even one-shot signals were played. Today, often multiple wavetables are used, and they are shaped with individual envelopes to create a time-varying signal. Wavetable synthesis is very well suited for acoustical instruments, but it is often too complex to synthesise instruments with time variable timbres.

Subtractive Synthesis

Subtractive synthesis starts out with a *basis waveform*, usually a periodic signal rich in partials, which is then filtered to achieve a usable timbre. Typically, the basis waveform and the filters both are modulated. Subtractive synthesis is well suited for the synthesis of original sounds, but it is hard to create accurate acoustic instrument simulations.

Additive Synthesis

Additive synthesis can be seen as the opposite to subtractive synthesis: simple sounds are combined to create more complex ones. Any periodic sound can be created by combining multiple sine waves (at different amplitudes, frequencies, and phases), and thus additive synthesis is one of the most versatile synthesis methods. The problem is that combining *many* sine waves, as required for real instruments, is very time consuming. It is mostly used in analysis – re-synthesis contexts.

Frequency Modulation Synthesis

Frequency modulation (FM) synthesis, first presented by Chowning (1973), makes use of the fact, that relatively rich spectra can be created by modulating the frequency of a sine oscillator by another. Often multiple oscillators are cascaded or coupled to create even more interesting sounds. FM synthesis is very easy to implement, but it is hard, or almost impossible, to predict the resulting sound of a determined configuration. It is therefore well suited for the synthesis of original sounds.

Physical Modelling

The term *physical modelling synthesis* is used for a group of synthesis methods that physically model musical instruments, or at least parts of instruments. For instance, we can build a mathematical model of the four strings of a violin and the resonance body, and then simulate the oscillations of the strings and the interaction with the resonance body. The resulting equations, typically differential equations, are normally cumbersome to solve and numerical approaches require a large amount of processing power. Nevertheless, with the increasing processing power of today's computing devices, physical modelling has become popular and it is well suited for wind and string instruments.

From our viewpoint, physical modelling has another importance: since there is a physical model of an instrument, it is possible to extend the model and incorporate gesture parameters as well.

2.5.3 Modalys

Modalys, formerly known as Mosaic (Morrison and Adrien, 1993), is a software package developed at IRCAM, which allows the design of virtual instruments based on simple physical objects such as strings, metal plates, tubes, plectra, reeds, and hammers. The users constructs an instrument and the defines how it will be played. Then, the resulting sound can be synthesised by using a physical modelling method called *modal synthesis*. Modal synthesis models a vibrating object by a bank of damped harmonic oscillators which are excited by an external stimulus. The frequencies and dampings of the oscillators are determined by the geometry and material properties of the object.

Since the user has the ability to design instruments based on simple objects, the software is particularly well suited for the integration of gestural concepts, such as piano mechanics and similar.

2.6 Human Hand Models

The mechanical complexity of the human hand is higher than any other part of the human body. The hand is able to perform fine motor manipulations and powerful work alike, and many dedicated parts co-operate in an highly optimised interplay to form a powerful union. Information on the anatomical units of the hand is found in (Putz and Pabst, 2001), and in detail in (Chase, 1990). The professional playing of musical instruments, and the piano in particular, puts demands on the hand unlike those of any other everyday or occupational human activity.

2.6.1 Hand Anatomy

The base of the palm consists of a number of small *carpal* bones (figure 2.3). The motions of these bones are mainly related to the flexion and abduction of the wrist. The shape of the *trapezium*, the carpal that articulates with the base of the thumb, has an enormous effect on the motion flexibility of the thumb. The remaining body of the palm is composed of four long bones called the *metacarpals* forming the base of each finger. They are connected to the carpals trough the carpo metacarpal (CMC) joints. The metacarpals for fingers 2 (index) and 3 (middle) are fixed, while the metacarpals for fingers 4 (ring) and 5 (pinky) are flexible in a small range. The metacarpal bone for the thumb is connected to the trapezium and is far more mobile than the others.

Each finger is composed of three additional bones, the *proximal*, *medial*, and *distal phalanges* (except for the thumb, which has no medial phalanx). The joint between the metacarpals and the proximal phalanges is the *metacarpal phalangeal* (MCP) joint, the next joint is the *proximal interphalangeal* (PIP) joint, and the last joint is the *distal interphalangeal* (DIP) joint. Except for the thumb, the MCP joints are more mobile than the PIP and DIP joints, which are just hinge joints. The CMC joint of the thumb makes it the most complex unit of the hand: the articular surfaces of this joint are both saddle shaped, and a loose, but strong, articular capsule joins the bones. Therefore the joint has an axis for flexion and extension as well as an axis for abduction and adduction. The looseness of the joint also allows for a small degree of passive rotary movement in the thumb. Putting all together, the hand has 27 *Degrees of Freedom* (DOF): four for each finger (three for flexion and extension, one for abduction and adduction), except for the thumb which has five DOF, leaving six DOF for the rotation and translation of the wrist.

2.6.2 Hand Models in Anatomy and Biomechanics

Beyond the field of computer science, important work has been done in anatomy and biomechanics. For example, in (Wagner, 1988) the hand size and joint mobility of professional pianists (127 male and 111 female) was studied. The work investigates in variability of hand sizes and of hand mobility, and discusses the tendency for greater mobility in pianists than in non-musicians. It concludes with the possibilities for practically using a biomechanical hand profile for the assessment of the manual aptitude of a pianist, and for using the available data for keyboard design.

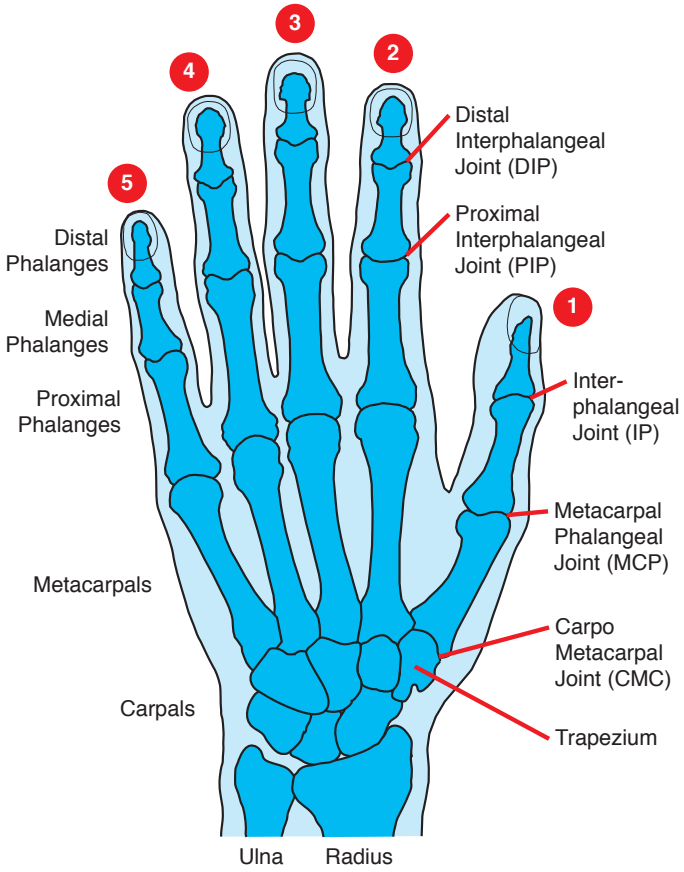

Figure 2.3: Skeleton of the human hand with annotated bones, joints, and finger numbers.

In (Buchholz et al., 1992) statistically-based anthropometry describing the kinematics of the human hand is collected. Then, an attempt is made to model this anthropometry by means of hand measurements, so it may be predicted non-invasively. The availability of anthropometric data is extremely valuable for modelling computer-based hand-models.

2.6.3 Hand Models in Computer Graphics

In face of the importance of our hands in daily life, but probably due to their immense complexity, human hand models have not received much attention in computer graphics. Only recently, realistic and sophisticated hand models have been developed (Albrecht et al., 2003; Bray et al., 2004) and many details are still work in progress. The availability of such models opens the doors to many applications, ranging from teaching and prac-

tising sign language (McDonald et al., 2001), or for teaching other manual skills, for instance music performance, which is a part of this thesis. Further, hand models are needed for interactive grasping in virtual environments, where visual feedback is required, and in simulation systems, e.g, for surgery planning. Finally, for realistic close-ups in computer graphics movies and computer games hand models with a high level of details and natural movements are desired.

Like most articulated figures, the human hand can be modelled as a collection of articulated rigid bodies connected by joints with one or more degrees of rotational freedom (Badler et al., 1993, 1999). Albrecht et al. (2003) presented a hand model with underlying anatomical structure and muscle contraction based animation control. The muscle contraction values are converted into values for a hybrid model, where pseudo muscles directly control the rotation of bones based on anatomical data and mechanical laws, while geometric muscles deform the skin tissue using a mass-spring system. Unfortunately, the computational cost of such models is still prohibitive for most real-time applications.

Hand models for playing musical instruments have also been presented. In (Kim et al., 2003) a system for the animation of the human hand playing violin was presented. The system consists of a neural network to control hand movement, and makes use of an optimisation method to generate examples for the neural network training. The musical decision for the correct finger use to press a string is made by best first search. Similarly, ElKoura and Singh (2003) presented a virtual guitar player with focus on correct string fingering for the guitar. They described a data driven algorithm to add sympathetic finger motion to arbitrarily animated hands. In addition a procedural algorithm generates the motion of the the fretting hand to play a given musical sequence on the guitar. What both models (or systems, respectively) have in common, is that they operate on a very low level of motion, higher level motion structures with respect to gesture and music have mostly been ignored.

As we shall see throughout this thesis, our work can be seen as a *pre-stage* to hand models in the computer graphics domain: our approach first considers musical structures and gestures, and then synthesises motion curves of finger tips, with the goal to correctly play the instrument from a musically and anatomically viewpoint. The resulting curves can then be fed into an arbitrary hand model, where other issues such as finger-joint motion, skin tissue generation, etc. are dealt with.

2.7 The Distributed Rubato Platform

Distributed Rubato (Müller, 2002) is a music research platform and is derived from *Classic Rubato* [1] (Mazzola and Zahorka, 1994a), which was originally developed on the NEXTSTEP platform, and has been ported to Mac OS X.[2] Distributed Rubato can be defined as a system of autonomous components called *Rubettes* with a unified communication interface. The Rubettes communicate trough that interface using a well-defined

[1] Sometimes, just "Rubato" is used for the same application. Here we use "Classic Rubato" to clearly distinguish from the Distributed Rubato platform.

[2] Classic Rubato is available for download at http://www.rubato.org.

and well-structured language. The type of application is not defined by the system itself, but by the Rubettes that are used to solve a certain problem. Thus, Distributed Rubato is not restricted to handle musical content. For instance, it has been used to process data delivered by geographic information systems (Rüetschi, 2001).

In contrast to Classic Rubato, which was written in Objective C and was restricted to the NeXT Platform, Distributed Rubato has been completely rewritten in Java, leading to a high degree of platform independence. Obviously this implies that the original Objective C code could not be used anymore, at least not directly. Nevertheless, we believed that the true value in Classic Rubato was in its methodologies and the implemented algorithms, which are usually ported from one language to another quite quickly. Observe that the terminology of "Rubettes" has been kept due to the similarity to their ancestors in Classic Rubato.

Rubettes communicate with each other using *Java Remote Method Invocation* (Java RMI) interface, the *Common Object Request Broker Architecture* (CORBA) or a similar remote object technology. Thus, they need not reside on the same machine – they can do their work on the computer of your neighbour, or on the server next to your office. The Internet makes it possible that they can even be distributed around the globe. Rubettes share a common language, which is a unified data representation and communication scheme: information exchange is exclusively based on the universal data model of Forms and Denotators (section 2.4). Again, Forms define named hierarchical spaces and Denotators are the points, or the substance in those spaces. This is a concept close to the one of XML, and in fact, data represented in XML can be translated to Forms and Denotators and vice versa. However, in contrast to XML, Forms and Denotators are directly related to algebraic structures. Thus they are available for algebraic operations in many cases.

Whether Rubettes are instantiated locally or on a remote machine is almost invisible to the user. Therefore, it becomes easy to move time- and memory-consuming tasks to another machine, where the needed resources are available.

The Distributed Rubato Framework provides the basic functionality that Rubettes need for operation. It can roughly be categorized as follows:

Distributed Rubato System Classes. They define the core classes that are needed to implement Rubettes and to launch the Distributed Rubato system. It further contains the communication framework which provides the seamless integration of remote Rubette objects.

Distributed Rubato Foundation Classes. This group contains classes that provides Rubettes with basic functionality for operation. This includes a large mathematics package with support for module calculus, arithmetics, matrices and linear algebra, parametric curves, and for logic-geometric operations on Denotators and Forms.

Rubette Repository. The Rubette Repository contains the Rubettes that have been implemented and that are ready for use. It also contains special Rubettes that are required for the operation of the whole system, such as an information agents, or visualisation and user interface modules.

Observe that the foundation classes are kept as general as possible. Application specific code is left to the Rubette implementations. Nevertheless, there are very general Rubettes that are not related to a particular application. Let us give two examples of such Rubettes. The first is the InfoRubette, which provides information about the Distributed Rubato infrastructure on a given machine. This particularly includes the information on what other Rubettes are available on that machine. Thus the InfoRubette must be instantiated on any machine where Distributed Rubato is running; otherwise communication with the outside world would be impossible. The second example is the PrimavistaBrowser (Mazzola and Göller, 2002), a Rubette very central to Distributed Rubato since it is responsible for the user interface of all interactive applications. It manages all multimedia representation and manipulation tasks and is able to display data in 3D – a necessity for its use in immersive environments such as caves. Other Rubettes do not provide user interfaces. They communicate with the browser trough the standard Rubette interface and provide the interactive data represented as forms and denotators. This mechanism is also used for the communication between several browsers, allowing direct collaboration of several users.

2.8 Soundium 2

Soundium 2 (Schubiger-Banz and Müller, 2003) brings together different areas of computer science in a unique multimedia system. Two main motivations were at the beginning of Soundium 2: first, the quest for a framework, which allows the user to test new ideas in audio processing and visualisation with minimal overhead. The realisation of new ideas requires a programmer-friendly framework with a well-designed programming interface, so the programmer can focus on the problem to be solved rather than struggling with the complex underlying system structure. Second, the system should be suited for interactive multimedia live performances (figure 2.4). Typical requirements behind this goal are short latencies and response times, real-time capabilities for audio-processing, and failure-free operation. The current client-server architecture in combination with data flow networks, signal processing, 3D graphics, versioned global state, and an advanced *Graphical User Interface* (GUI), manage to match these goals.

Many of the concepts of Soundium 2 are well-known, the system incorporates features that have been available in real-time audio-processing systems, such as the Max family (Pukette, 2002), a visual programming system for signal processing, including an extension for video processing (Matsuda and Rai, 2000). Another example is SuperCollider (McCartney, 2002) a computer music programming language with integrated processing and process description.

2.8.1 System Architecture

Figure 2.5 depicts the overall architecture of Soundium 2. Basically, Soundium 2 is based on a multi-tier client/server architecture with one client controlling a cluster of servers called “engines”. The motivation for this separation is twofold: the first motivation is

Figure 2.4: Live video projections produced by Soundium 2.

code complexity and code maintenance and the second is timing. The engines should handle all timing sensitive operations while keeping their code complexity small. All complex operations should run on the client, which may or may not control the engines in real-time. This allows us to rapidly develop and fine-tune individual engines for specific tasks such as 3D rendering or audio processing while performing more complex tasks like global-state and GUI management in the client.

The lowest architectural level is represented by specialised *hardware* (hardware level) such as video projectors, MIDI equipment, or environmental sensors. All these devices are controlled by individual *engines* (engine level) which are optimized for a specific task. Each engine holds a part of the global dataflow network state which is manipulated through an *Remote Procedure Call* (RPC) interface. This RPC interface is mainly used by the *client* (soundium level) to configure and parameterize the engines. Only the client holds the entire global system state composed of the per-engine dataflow networks as well as the system wide state modification history. The client also renders a multi-user graphical user interface (GUI level) that can be accessed via the *Remote Frame Buffer* (RFB) protocol (Richardson and Wood, 1998) over the network for collaborative performances.

Graphical User Interface

The Soundium 2 graphical user interface allows interactive inspection and manipulation of the system state as well as editing SL1 code. Figure 2.6 shows a screenshot of the GUI with a selection of user interface elements. The *system state* represents the union of all dataflow networks on each engine as a graph. *SL1 scripting* is used for modifications of

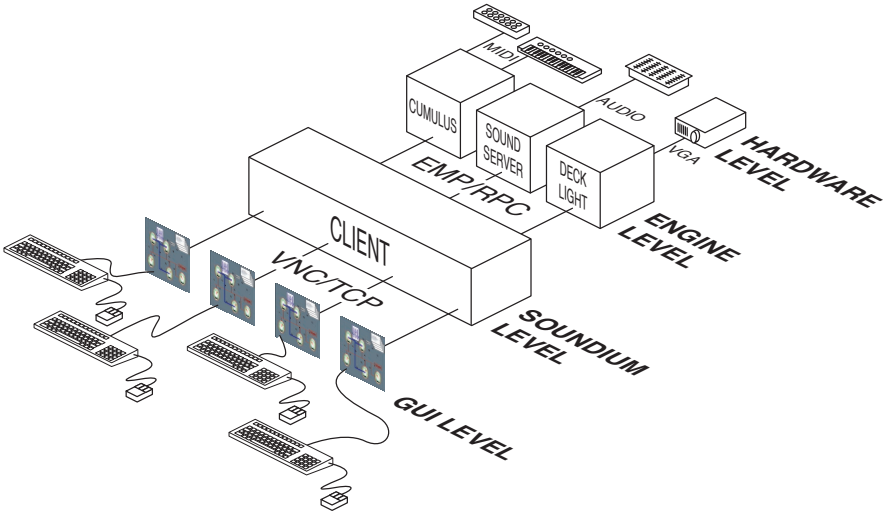

Figure 2.5: Overview of the Soundium 2 architecture.

SL1 code. The *revision tree* holds the trace of all state changes in the system and allows arbitrary jumps between system states. The revision tree can also be seen as a multi-level redo/undo tree. Node *parameters* can either be modified through dedicated GUI elements or associated with hardware devices such as *MIDI controllers*.

Client Side Scripting

The global system state (the union of the per-engine dataflow networks) is versioned and stored as a revision tree. Every state manipulation is saved in the revision tree and it is thus possible to rollback to any previous state and starting a new branch from there. Even forward propagation of state changes is supported if the changes are non-conflicting with later revisions of that state. The integration of versioning in a multimedia system ensures that no artwork is lost and that new artwork can be easily evolved from existing work. The storage format is called Soundium Language 1 (SL1), a full-fledged imperative programming language. Besides encoding the system state, SL1 is also used for client side scripting, GUI customization, and performance preparation.

Automatic Software Configuration

Resource classification is the foundation for the automatic software configuration (Schubiger, 2002) capability used in Soundium 2. By classification we understand the process of determining resource properties such as content type as well as extracting features like beat positions. A modular scheme allows the addition of resource transformations (such as format conversion), which are automatically aggregated based on their semantics in order to construct high-level transformations. Software configuration relies on a formal

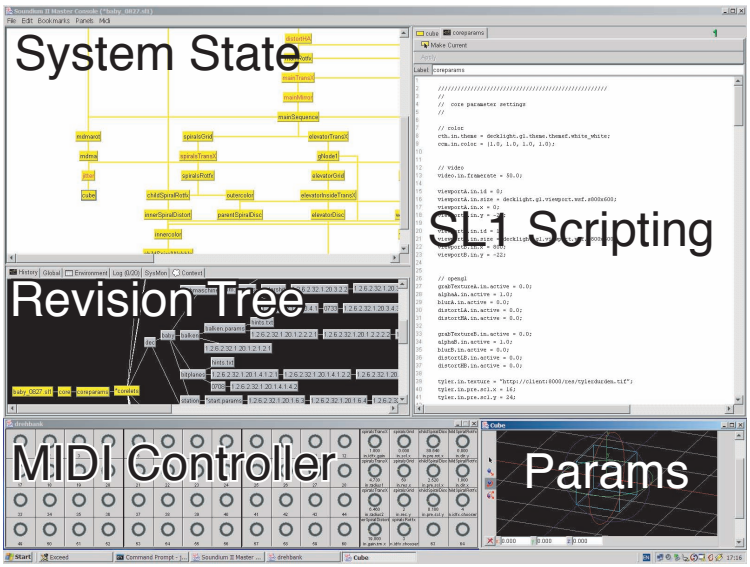

Figure 2.6: The Soundium 2 graphical user interface.

ontology, which is planned to include meta information beside the current media related information such as music style or mood.

Resource Handling

The management of resources such as 3D models, textures, and audio clips is entirely HTTP (Hyper Text Transfer Protocol) based. In addition to flat file resources, a proprietary extended resource format (XRS) allows association of SL1 code with media files, thereby transforming passive resources into active objects that can react to events like resource loading.

The Remote Procedure Call Interface

The RPC interface is extremely simple, basically allowing the manipulation of the engine's data-flow network. The calls comprise creation and removal of nodes as well as edges and reading and writing of parameters. Every call takes a timestamp argument occurring in the future, which is used by the engine to properly schedule the execution of the call. Two auxiliary calls allow obtaining static engine information (supported node types and data formats) and querying the current engine time.

Engine

The Soundium 2 engine is a C++ framework based on a processing graph of building blocks (the nodes) that communicate which each other (along the graph edges). It has been designed with two main goals in mind: First, provide efficient real-time processing facilities to different media types. Second, provide a highly abstracted programming interface for the building blocks that allows non-experts to write their own signal- and media processing code.

The first goal has been achieved by implementing a system that makes use of extensive multithreading. The building blocks are automatically assigned to *schedulers*, which handle processing characteristics of a specific media type. Data-flow analysis determines the execution order inside the different scheduling groups. The building blocks communicate trough connected ports allowing arbitrary data types. Inter-scheduler communication is automatically synchronised.

The application of rigorous object-oriented methods in C++ almost completely hides scheduling and synchronisation issues from programmers of building blocks. For instance, ports can be accessed like normal variables. To date, building blocks for sequencing, audio signal processing, MIDI I/O, parameterized OpenGL rendering graphs, and OpenGL Performer have been implemented.

Chapter 3

Gesture Curves and Gesture Spaces

This chapter deals in detail with our model of musical gestures where gestures are represented by high-dimensional parametric curves residing in arbitrary symbolic or physical spaces. Having the same model for different spaces or realities was one of our core motivations since this opens the path to a unified theory of dealing with gestures in general. The model has first been presented by Mazzola (2002b), and its evolution has been published in (Müller, 2002, 2003; Mazzola and Müller, 2003).

We will first introduce our concept of *gesture curves* residing in *gesture spaces* and define corresponding transformations between them. The concept of gesture spaces is derived from the concepts of score and performance spaces and the performance transformation $\wp$, well known from performance theory (section 2.3). Will will also introduce *lifted pairs of spaces*, which express the vertical relationship between score (or performance) spaces and gesture spaces.

The second section deals with operations on basic gesture curves and will show how the composition of such curves can be used to create gesture curves (and therefore gestures) at the next higher conceptual level. While we are not able to present a complete grammar of musical gestures in this work, we believe that providing a basic alphabet and appropriate composition mechanisms will eventually yield the basis of such a grammar.

The key to computational methods for dealing with gesture curves is given in sections 3.3 and 3.4. The theory deals with the algorithmic construction and manipulation of gesture curves. We shall particularly focus on the construction of gesture curves from musical scores and performance scores and the constraint-based shaping of physical curves with respect to a kinematics-based hand model for keyboard instruments.

3.1 A Model for Musical Gestures

Let us first define a mathematical construct that is able to represent a musical gesture (or any gesture):

Definition 3.1 *Let* F

$$F \longrightarrow \textbf{Limit}(F_1, \ldots, F_n), \tag{3.1}$$

be the Gesture Form *over the parameter Forms* F_i*:*

$$F_i \longrightarrow \textbf{Simple}(\mathbb{R}), i = 1, \ldots, n. \tag{3.2}$$

Work over the topos $Top^{@}$ *instead of* $Mod^{@}$*, where* Top *is the category of topological spaces together with continuous maps, e.g.,* $I = [0, 1] \subset \mathbb{R}$*,* $Top(I, \mathbb{R}) = I@\mathbb{R} = \{f : I \to \mathbb{R}, continuous\}$.

A gesture G *is a Denotator in* F *at address* I*, i.e.*

$$G : I \rightsquigarrow F(\gamma) \tag{3.3}$$

with $\gamma \in I@F$*. Thus, the parametric* gesture curve γ *maps from* I *to* $Space(F)$*:*

$$\gamma : I \to Space(F) = \mathbb{R}^n \tag{3.4}$$

with $Space(F) = \oplus_{i=1}^{n} Space(F_i) = \mathbb{R}^n$.

This definition was chosen for the following reasons: a) parametric curves are open to computational methods; b) we can define operations on curves, e.g., products, or concatenation of curves; and c) the curves are well-suited for physical models, where movements of points in a physical space are modelled. Observe that in contrast to dealing with musical scores, using topos $Top@$ implies topological and not algebraic structures in this framework.

We clearly point out that the curve parameter t is not necessarily related to time (be it a symbolic or physical time). Gesture curves can reside in spaces where time does not appear at all. However, a gesture representing a movement in the physical domain, such as movement of a finger, implicitly involves physical time, which then is also a function of t.

As an illustration, figure 3.1 shows a gesture curve in a simple vector space $F = (e, a, b, c)$. Here, we have the typical set-up that the time parameter e (physical onset time) is also a function of t. In an abstract context, the parameters a, b, and c are arbitrary. In contrast, in a physical context, they would be replaced by space dimensions (i.e., x, y, z), or other physical parameters (e.g., forces or velocities), or any appropriate combination.

Figure 3.2 shows a symbolic gesture curve G, which models a (monophonic) gesture of a keyboard-like instrument. The curve is closely modelled after MIDI concepts ("Note on", "Note off", and "Velocity", the derivative of the figure's position coordinate). We shall later see that this simple model can be used to construct more complex and polyphonic gesture curves for keyboard instruments.

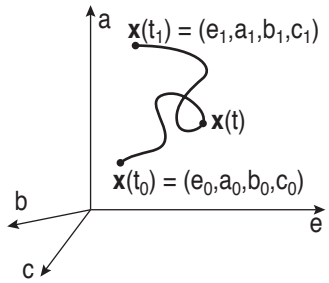

Figure 3.1: A gesture curve in a generic setup. The curve is parameterised in $t \in [0, 1]$.

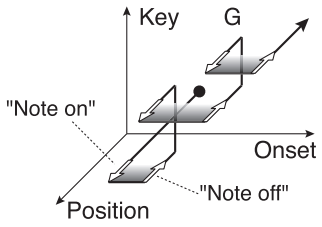

Figure 3.2: A symbolic gesture curve for a MIDI keyboard instrument.

3.1.1 Music Notation and Gestures: A Thesis

So far, we have not dealt with the relationship of musical scores and musical gestures, and our model of gestures, i.e., the gesture curves, in particular. Musical scores provide an abstract way of writing down musical compositions. Musical gestures, on the other hand, exist in a vast range from an abstract, mental space to very explicit and concrete movements in a physical space, such as curves denoting the position of finger tips playing an instrument. In order to link the two realities, Mazzola (2002b) has raised the following thesis:

Thesis 3.1 *Symbols in a musical score (e.g., the notes) can be seen as "frozen" gestures.*

This thesis is supported by the observation, that today's music notation originated from *neumes*. As already mentioned in section 2.2 neumes are an early form of music notation (Parrish, 1957), and the word "neuma" is actually the Greek word for "hint". Symbolic music notation can therefore be seen as a highly abstract way of writing down gestures (refer to figure 3.3 for a simplified schema showing the increasing abstraction from neumes to today's notation).

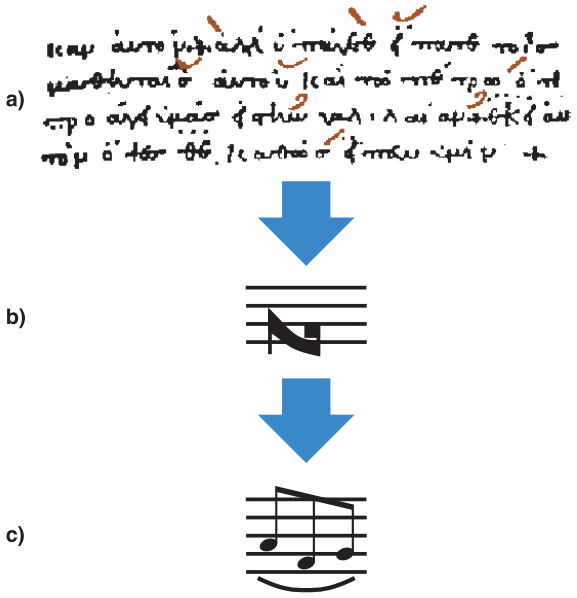

Figure 3.3: Evolution of music notation: From neumes (a) via square notation (b) to music notation as it is used today (c).

3.1.2 Lifted Pairs of Spaces and Gesture Transformations

The relationship given in the previous section can now be put together with the relationship given by a score S, a possible performance P, and a performance transformation $\wp$ between the two, as we know it from performance theory. This combination is presented in figure 3.4: the lower part of the figure shows the score space S and the performance space P containing note and tone events, starting at the position of the black dots and having the duration of the length of the line. The performance transformation is now denoted as $\wp_{Score}$. Analogous, a *symbolic gesture space* and a *physical gesture space* have been added in the upper part of the figure, and, between the two, a *gesture transformation* $\wp_{Gesture}$. The complete situation yields the following definition:

Definition 3.2 *Let S be a musical score space and P a performance space, with a performance transformation $\wp_{Score}$. Then the symbolic gesture space SS together with S, and the physical gesture space PP together with P are called* lifted pairs of spaces. *The vertical operations are called "freezing" when transforming from SS to S (and from PP to P, respectively), and "thawing" when transforming from S to SS (and from P to PP).*

In other words, the "freezing" and "thawing" operations express the vertical relationships between score (or performance) spaces and symbolic (or physical) gesture spaces. We shall see later, how these operations can be realised.

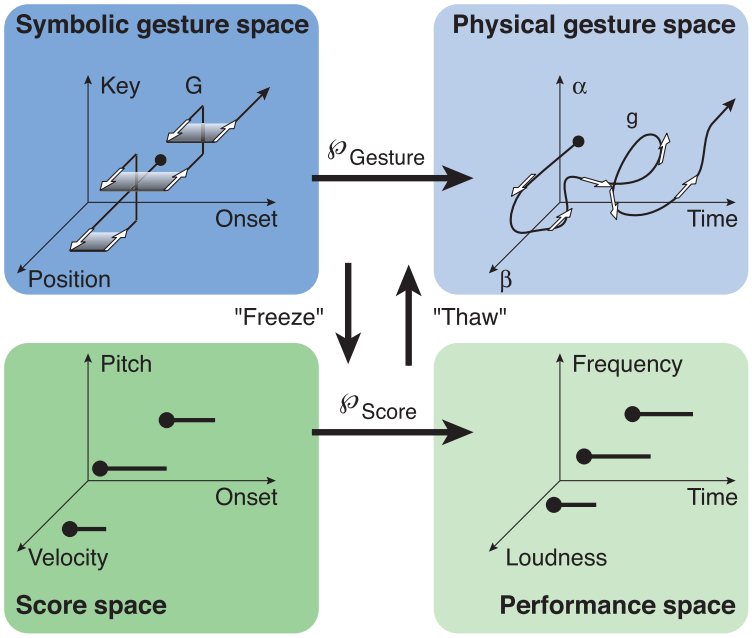

Figure 3.4: Gesture spaces containing the symbolic gesture curve G and the physical gesture curve g, their inter-relationships, and the relationship to the corresponding instrument spaces.

The physical gesture curve $g = \wp_{Gesture}(G)$, on the top right in figure 3.4 represents the transformed symbolic gesture curve G. Here, the parameters are of geometric nature, such as angles between finger segments, and motion parameters, such as velocity and acceleration vectors of the finger tips or of the ankles. Those parameters are represented by the figure's α and β axes.

3.2 Operations on Gesture Curves

An important issue is the question of how complex gestures can be built from simpler ones. This section defines basic mathematical operations on gesture curves. These operations can be seen as a vocabulary of useful manipulations and transformations of curves which will be needed for the construction and shaping processes in the sections later on.

3.2.1 Constant Gestures

First, we start out with the most basic of all possible gestures, the constant gesture G_{const}:

$$G_{const} : I \rightsquigarrow F(\gamma_{const})$$

with $\gamma_{const}(t) = c = const \in Space(F)$ for all $t \in I$.

3.2.2 Add and Scale

The space $I@F$ of gestures becomes a real vector space by defining add and scale operations. Assume two gesture curves $\gamma_1, \gamma_2 \in Space(F)$. Then we can define the addition of γ_1 and γ_2:

$$(\gamma_1 + \gamma_2)(t) = \gamma_1(t) + \gamma_2(t) \tag{3.5}$$

The scaling of a gesture curve $\gamma \in Space(F)$ by a constant $r \in \mathbb{R}$ can be defined as

$$(r \cdot \gamma)(t) = r \cdot \gamma(t) \tag{3.6}$$

As a special case we define the shift operation as an addition of a gesture curve $\gamma \in Space(F)$ with a constant gesture curve $\gamma_{const} \in Space(F)$:

$$(\gamma + \gamma_{const})(t) = \gamma(t) + \gamma_{const} \tag{3.7}$$

3.2.3 Reverse Operation (Switch)

Gestures can be reversed, this is achieved by reversing the ordering of the parameter $t \in I$: Let $\rho : I \to I : t \mapsto 1 - t$, then γ^ρ is the *reverse gesture curve* of γ, with $\gamma^\rho = \gamma \cdot \rho : I_\rho \xrightarrow{\rho} I_\gamma \xrightarrow{\gamma} Space(F)$.

3.2.4 Concatenation

The concatenation of two gesture curves $\gamma_1, \gamma_2 : I \to Space(F)$ provides the foundation for building more complex gestures out of simpler ones. Condition for the ability to concatenate γ_1 and γ_2 is that $\gamma_1(1) = \gamma_2(0)$. Then the concatenation is defined as:

$$\gamma(t) = \gamma_1(t) \propto \gamma_2(t) = \begin{cases} \gamma_1(2t) & \text{if } 0 \leq t \leq 0.5 \\ \gamma_2(2t-1) & \text{if } 0.5 < t \leq 1 \end{cases} \tag{3.8}$$

3.2.5 Product

Let $\gamma \in Space(F)$ and $\delta \in Space(G)$, with $F \longrightarrow$ **Limit**$(F_1 \dots F_n)$, and $G \longrightarrow$ **Limit**$(G_1 \dots G_m)$. Then we can define the *product gesture* curve $\gamma \times \delta \in I@(F \times G)$

$$\gamma \times \delta(t) = (\gamma(t), \delta(t)) \in \mathbb{R}^{n+m} \tag{3.9}$$

with $F \times G \longrightarrow$ **Limit**$(F_1 \dots F_n, G_1 \dots G_m)$.

3.2.6 Top Space

Finally, a *gesture of gestures* can be constructed by using the top space of the real vector space $F_I := I@F$, $F_{Gestures} \longrightarrow$ **Simple**(F_I). Then, the 'gesture curve' $\gamma : I \to Space(F_{Gestures})$ is a parameterised curve of gestures.

3.3 Initial Construction of Gesture Curves

Before discussing the construction of complex, constrained gesture curves (typically in the physical domain), in this section, we deal with the "thawing" operation of simple, unconstrained symbolic gesture curves. The curves are considered unconstrained because we are able to construct them in a symbolic reality where for instance hand anatomy, or physical laws are of no importance. The curves built by that method live on their own rights in a symbolic reality.

For the discussion of the "thawing" operation, consider the piano-roll like scores in figure 3.5. Pitch is given by the vertical axis, onset time by the horizontal axis. The four events reside inside the onset boundaries ε_0 to ε_5. The lower score has been annotated with fingering information, which we assume to be given, either by manual definition or by an automatic estimation (Parncutt, 1995).

One of the main problems with symbolic gesture curves is the issue that fingers have to move at infinite speed in some cases: for instance at ε_3 the second event for finger 2 ends and at the same time as finger 3 has to start playing the third event. In addition, at ε_4 finger 3 still playing the third event, or more precisely, is just about to release the key, and has to start playing the fourth event. Thus, there is in fact no time left for moving the finger from event three to event four. This problem has been solved by parameterising onset time for each symbolic finger, i.e., onset time E becomes a function of the curve parameter t: During the transition between the two events, the position co-ordinate increases, while onset time remains constant. Thus, in a symbolic gesture curve, fingers are allowed to move at "infinite" speed.

For the construction of the curve, each finger is handled separately. First, the curve parameter t is divided by the number of events for each finger. Then, each event is divided into three intervals, one for the transition before the event takes place, one for the event itself, and one for the transition after the event. Finally, cubic interpolation is applied for each subinterval. The interpolation type is however not part of the intrinsic definition of a symbolic gesture curve.

Figure 3.6 (a) shows the symbolic gesture curve for finger 2. Each axis is drawn separately in function of curve parameter t. The semi-transparent vertical bar denotes the area where the actual event takes place. As we just have seen, onset time E also exists in the gesture space, but separated for each finger. The remaining instrument parameters H, L, etc., are replaced by pseudo space co-ordinates X, Y, Z, which define the co-ordinate system for a virtual keyboard. For the moment, we do not care about the units or the values of those space co-ordinates, it is enough to assume that they represent some geometric 3D space. The effective mapping functions from symbolic score parameters to a physical instrument space will be given in section 3.4.4. X_2 is the position on the keyboard and corresponds to pitch, Y_2 is the position above the keyboard and tells whether key is pressed or not, and the derivative $Y_2' = dY_2/dt$ contains information about the speed at which the key is pressed or released, respectively. This speed corresponds to the loudness of a certain event. Note that the Z position (depth on the keyboard, thus defining a white or a black key) is omitted in the figure. Its construction is analogous to the one of X. Figure 3.6 (b) shows the symbolic gesture curve for finger 3. Indicated by the dashed

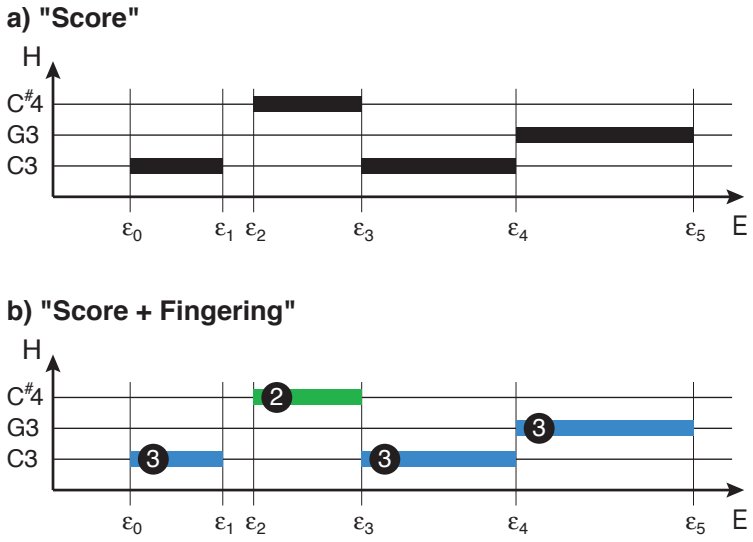

Figure 3.5: A symbolic score, without (a), and with (b) annotated fingering information. Horizontal axis denotes onset time, vertical axis pitch.

square is the region where onset time E remains constant since finger 3 has to move from event 3 to event 4 at infinite speed.

3.4 Constrained Shaping of Gesture Curves

The previous section has dealt with the construction of unconstrained gesture curves. The method can be used for the construction of symbolic gesture curves based on a musical score. However, the attempt to construct physical gesture curves from performance scores imposes new requirements. For instance, hand anatomy has to be taken into account: we have to deal with maximum distances between finger tips, or overlapping of hand parts has to be avoided. Further, we need to take care of dynamic facts: fingers can not move at infinite speed anymore, their acceleration and velocities are limited, depending on the maximum forces and the finger's mass, according to Newton's law.

In this section, we will deal with the problem of the construction and the shaping of physical gesture curves, which then can be represented by virtual performers or by body parts of virtual performers. Unfortunately, we do not know anything about a possible direct transformation $\wp_{Gesture}$ which was given in figure 3.4. Thus, instead of defining $\wp_{Gesture}$, we begin with a given symbolic gesture curve, which can for instance be obtained directly from a given MIDI file, and then "freeze" the curve (figure 3.7, step 1), which yields events in the score space. From here, $\wp_{Score}$ can be applied (figure 3.7, step 2), as it has been done in the past (Mazzola, 2002c): $\wp_{Score}$ is defined by performance vector fields, which are numeric results delivered by a number of analyses (e.g.

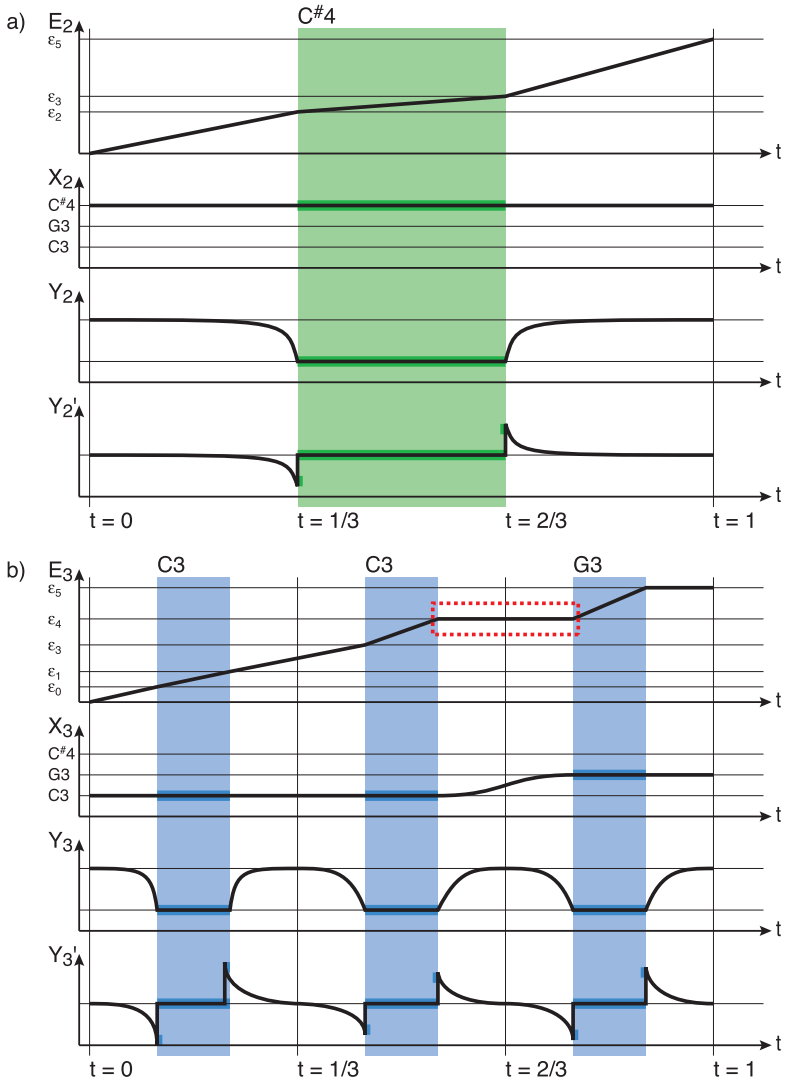

Figure 3.6: Independent symbolic gesture curve for fingers 2 (a) and 3 (b), with curve parameter t running from 0 to 1 on the horizontal axis.

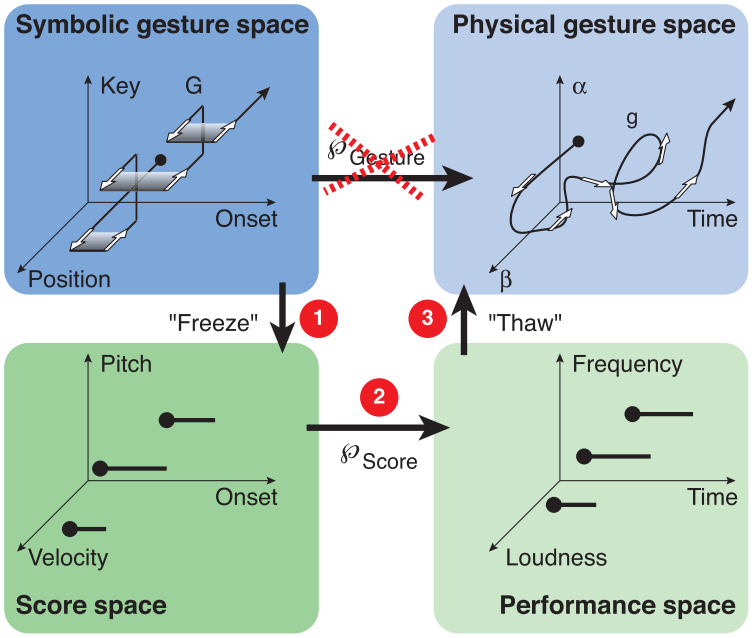

Figure 3.7: The performance of a symbolic gesture curve. Instead of applying the direct transformation $\wp_{Gesture}$, the gesture curve G is first "frozen" (step 1), then $\wp_{Score}$ is applied (step 2), and finally the physical gesture curve g is "thawed" from the performance score.

melodic, harmonic, motivic). The process results in physical sound events in the performance space. What remains, is to "thaw" those events (figure 3.7, step 3), resulting in the physical gesture curve we are looking for. Thus, this section deals with the rather complex "thawing" operation in the physical domain, i.e., the construction of physical gesture curves.

As we shall see now, the concepts of construction of symbolic gesture curves will also be helpful in the physical domain: assume the score in figure 3.5 to be a score that has already been performed, e.g., some MIDI recording. Then, the application of the curve construction algorithms from the former sections yields the symbolic curves given in figure 3.6. However, the construction algorithm will be slightly modified in order to make the application of anatomic constraints easier.

As a next step we will define a constrained hand model whose constraints can be written as a number of equalities and inequalities. These constraints will have to be solved for every segment in the score. By making use of Sturm's theorem for cubic splines, there is a general approach for finding solutions. The large number of parameters and the resulting polynoms of up to the 12th degree, however, make it almost impossible to find a symbolic solution to the problem as our experiments with mathematics packages such as

Mathematica have shown. Thus, it is helpful to split the problem in two parts: first, solve the anatomic constraints and then solve the physical constraints by shaping the time axis using the Sturm theorem for the one-dimensional case.

Splitting the problem is supported by an observation that comes from rehearsal: when a pianist rehearses a sequence of a piece, he first starts playing the sequence slowly until the finger movements are correct and internalised. Then he can begin playing the sequence towards the required speed – without elementary changes to the geometric shape of the gesture.

3.4.1 Curve Construction Revisited

A first approach would construct multiple curves according to section 3.3 and then further subdivide and reshape the curves where required. However, it is preferable to follow the approach of starting out with basic, well-defined curve segments, which then can be joined to build more complex curves (in the sense of section 3.2) for the following two reasons: first, unconstrained construction results in curves with individual time segments. When it comes to implementation, numerous mappings between t and time have to be calculated, making the reshaping process inefficient. Second, and more important, is the point of view that every gesture is related to other gestures in a constrained situation. This is not only true in the physical domain, but also in symbolic domain: for instance, left hand and right hand playing in piano performance can never be completely independent. Left-hand and right-hand gestures are always mentally and physically linked.

The above observations allow us to revise the construction mechanism for symbolic gesture curves based on a score. Again, each finger is handled separately, but in contrast to section 3.3, for each finger, the onset axis E is divided into the same number of segments. Initially, the mapping from curve parameter t to onset time E is kept identical for each finger. Thus, we have full control of a finger's motion in an interval $[t_0, t_1]$ with respect to the other fingers' motions. Again, the transition before the event, the event itself, and the transition after the event are handled separately. In addition, and in contrast to the former construction method, segments unrelated to a musical event will appear. An example would be a chord of three notes, where two fingers are not involved in playing, but nevertheless need to be in correct position. To summarise, the types of appearing score segments can be categorised into *None*, *Attack*, *Hold*, and *Release*. The individual segments are created according to the rules given in figure 3.8. At the beginning and at the end of the score Hold segments are added, these ensure enough room for gestures before and after the score's events are involved. Figure 3.9 shows the (unshaped) symbolic gesture curves that have been constructed according to above mechanisms. For each finger, the curve is divided into 17 segments. The type of segments are denoted by 'N' for None, 'A' for Attack, 'H' for Hold, and 'R' for Release.

3.4.2 Defining the Virtual Keyboard

Obviously, the dimensions and structure of the keyboard are reflected in some of the constraints to be defined. Defining a virtual keyboard is fairly easy, since all dimensions are

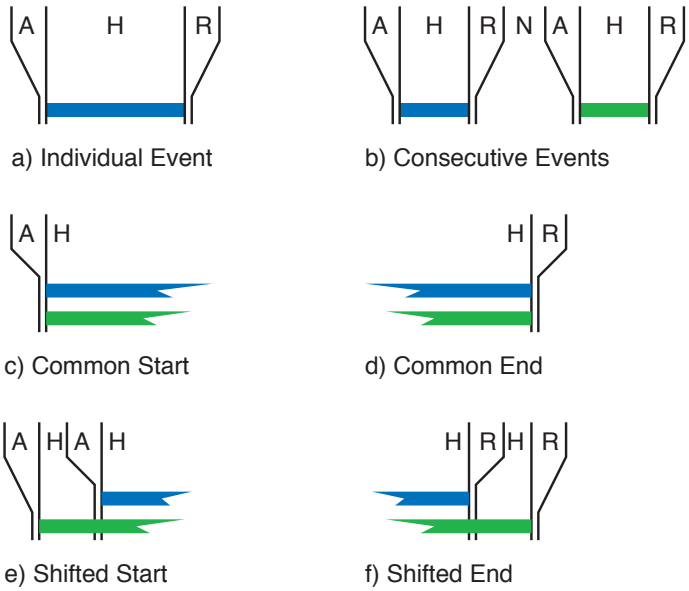

Figure 3.8: Score segmentation rules for different event combinations. Horizontal axis denotes onset time, vertical axis denotes pitch. The uppercase letters denote the segment type: N for None, A for Attack, H for Hold, and R for Release.

well-defined in standards (DIN8996, 1985). Figure 3.10 shows a top and a front view of some keys of a typical keyboard with annotated dimensions. The co-ordinate origin is placed in front of the C3 key, and the co-ordinate set-up corresponds to the co-ordinate system given for symbolic gesture curves in earlier sections: the x co-ordinate corresponds to pitch, the y co-ordinate to the position above the keys, and the (negative) z co-ordinate to the position on the key itself.

3.4.3 Defining a Constrained Hand Model

The next step is to define a hand model which is well-suited for piano performance. The hand model in our approach focuses on the movements of the finger tips and of the hand root, since these movements are the most essential ones in a piano performance. The resulting motion curves can be used to animate a general hand model, such as the one in (Bray et al., 2004). Therefore, and as mentioned earlier, our model can be seen as a computational pre-stage, whose function is to focus on the *gestures*, which represent not only a determined performance, but also playing style and technique. All other issues will be covered by the general hand model. These include appropriate handling and representation of the hand skeleton, the muscles, and the skin.

Thus, for each hand we define six gesture curves, $\gamma_{root}(t)$ for the hand root, and $\gamma_i(t),\ i = 1, \ldots, 5$ for each finger. This situation is illustrated in Figure 3.11. The collec-

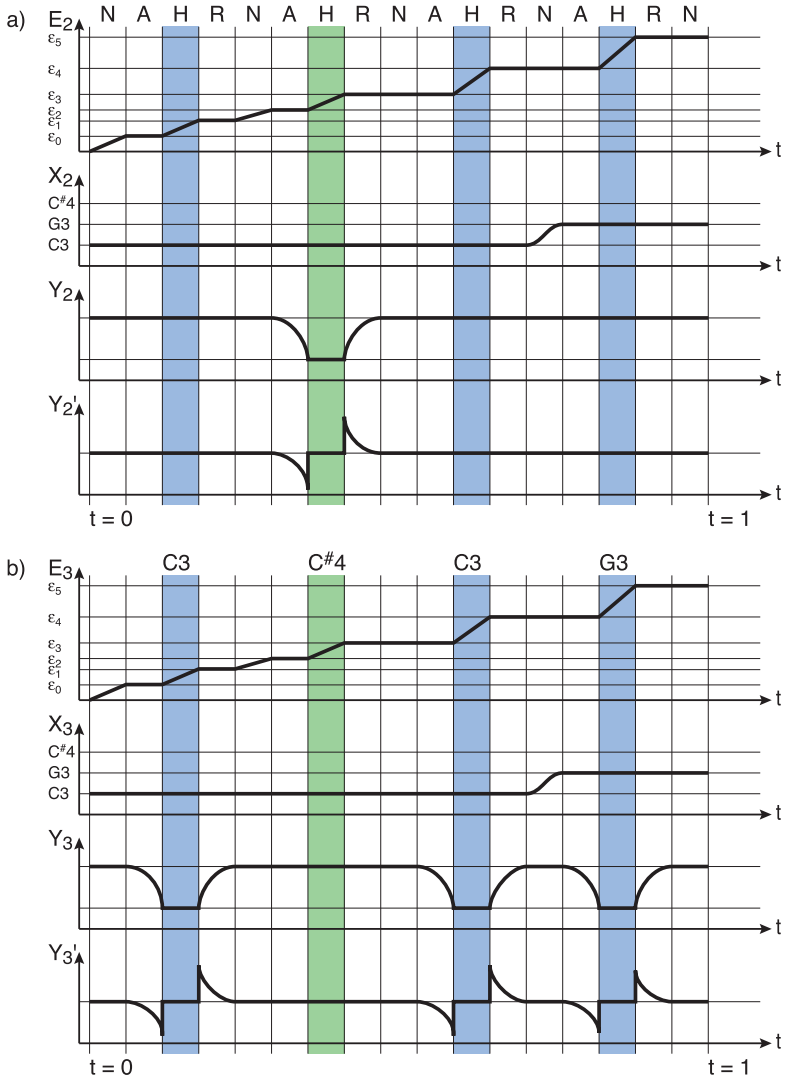

Figure 3.9: Dependent symbolic gesture curve for fingers 2 (a) and 3 (b), with curve parameter t running from 0 to 1 on the horizontal axis. The uppercase letters denote the segment type: N for None, A for Attack, H for Hold, and R for Release.

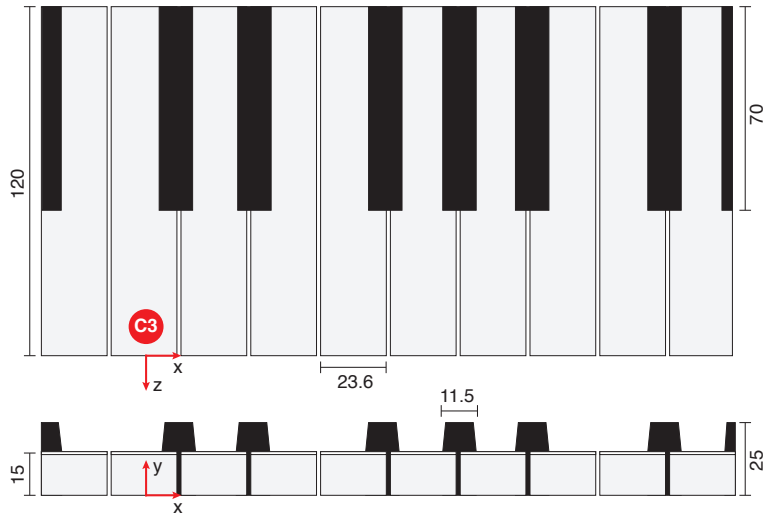

Figure 3.10: Top and front view of standard keyboard layout with co-ordinate origin at C3 and key dimensions after DIN8996.

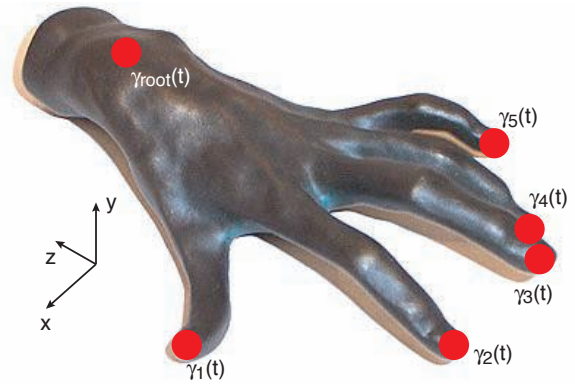

Figure 3.11: Hand model with annotated positions for gesture curves $\gamma_{root}(t)$ and $\gamma_1, \ldots, \gamma_5$

tion

$$\gamma = (\gamma_{root}, \gamma_1, \ldots, \gamma_5) \tag{3.10}$$

is the curve for the whole hand, and we are looking for the evaluation of γ at a parameter value t:

$$\gamma(t) = (\gamma_{root}(t), \gamma_1(t), \ldots, \gamma_5(t)) \tag{3.11}$$

For each gesture curve $\gamma_{root}(t), \gamma_1(t), \ldots \gamma_5(t)$ we use separate co-ordinate functions $x_i(t)$, $y_i(t)$, $z_i(t)$ and a common onset time function $e(t)$:

$$\gamma_{root}(t) = (x_{root}(t), y_{root}(t), z_{root}(t), e(t)) \tag{3.12}$$

$$\gamma_i(t) = (x_i(t), y_i(t), z_i(t), e(t)),\ i = 1, \ldots, 5 \tag{3.13}$$

The according space functions (space co-ordinates without onset time function) are defined as:

$$\gamma_{root}^{Space}(t) = (x_{root}(t), y_{root}(t), z_{root}(t)) \tag{3.14}$$

$$\gamma_i^{Space}(t) = (x_i(t), y_i(t), z_i(t)),\ i = 1, \ldots, 5 \tag{3.15}$$

Our approach then uses cubic polynomial functions as gesture co-ordinates, i.e., 76 coefficient variables:

$$x_i(t) = x_{i,3}t^3 + x_{i,2}t^2 + x_{i,1}t + x_{i,0} \tag{3.16}$$

$$y_i(t) = y_{i,3}t^3 + y_{i,2}t^2 + y_{i,1}t + y_{i,0} \tag{3.17}$$

$$z_i(t) = z_{i,3}t^3 + z_{i,2}t^2 + z_{i,1}t + z_{i,0} \tag{3.18}$$

$$e(t) = e_3t^3 + e_2t^2 + e_1t + e_0 \tag{3.19}$$

Above equations define the movement of the hand for a given segment in the score. The next step is to define all required constraints. The equations together with the constraints will result in a number of equalities and inequalities. The goal then is to find solutions for the 76 coefficient variables.

Boundary Conditions

The first group of constraints are boundary conditions for all five fingers. These include a) *positional equations*, and b) *velocity equations* that are given by the velocity of a note to be played. The positional equations are given as constants at the beginning and the end of the curve for each finger $i = 1, \ldots, 5$:

$$\gamma_i(0) = \gamma_{i,0} \tag{3.20}$$

$$\gamma_i(1) = \gamma_{i,1} \tag{3.21}$$

The velocity equations need to be defined *only* for those fingers involved in a attack or a release process. We start out with derivatives with respect to the curve parameter t:

$$D\gamma_i^{Space} = \frac{d\gamma_i^{Space}}{dt}$$

$$De = \frac{de}{dt}$$

Consequently, the velocity equations are

$$\frac{D\gamma_i^{Space}(0)}{De(0)} = D\gamma_{i,0}^{Space} \tag{3.22}$$

$$\frac{D\gamma_i^{Space}(1)}{De(1)} = D\gamma_{i,1}^{Space} \tag{3.23}$$

where $D\gamma_{i,0}^{Space}$ and $D\gamma_{i,1}^{Space}$ are given border velocity parameters at $t = 0$ and $t = 1$, respectively.

Boundary Inequalities

The second group of constraints are boundary inequalities that result from the hand model: we want the hand root and the finger tips to remain inside given boundaries in order to make sure that the hand remains in a valid anatomic state. First, we define boundary boxes for the hand root and each finger. It is noteworthy to say that using simple boxes does not reflect all possible states of a real hand. But defining the boxes small enough, i.e., as a subset of all reachable positions, can be considered as being appropriate for a realistic piano performance. The hand model and the 6 bounding boxes are illustrated in figure 3.12. The size of the boxes is only given as qualitative dimensions.

Assume that we have defined Box_{root} for the hand root, and Box_i for each finger $i = 1, \ldots, 5$. First, we can define the boundary conditions for the hand root as an absolute box position:

$$\gamma_{root} \in Box_{root} \tag{3.24}$$

Then, we define the boundary conditions for the thumb and the index finger as relative box positions to the hand root:

$$\gamma_1^{Space} - \gamma_{root}^{Space} \in Box_1 \tag{3.25}$$

$$\gamma_2^{Space} - \gamma_{root}^{Space} \in Box_2 \tag{3.26}$$

Finally, we define the boundary conditions for the remaining fingers as relative box positions to the index finger, and subsequently to each other. This makes sure that we can avoid collisions between finger tips:

$$\gamma_3^{Space} - \gamma_2^{Space} \in Box_3 \tag{3.27}$$

$$\gamma_4^{Space} - \gamma_3^{Space} \in Box_4 \tag{3.28}$$

$$\gamma_5^{Space} - \gamma_4^{Space} \in Box_5 \tag{3.29}$$

In addition, we need to assure the non-intersection of the thumb with the other fingers. Therefore consider the two finger bases b_2 and b_5, as illustrated in figure 3.13. Suppose that we have two *constant difference vectors* d_2 and d_5 such that:

$$b_2 = \gamma_{root}^{Space} + d_2$$

$$b_5 = \gamma_{root}^{Space} + d_5$$

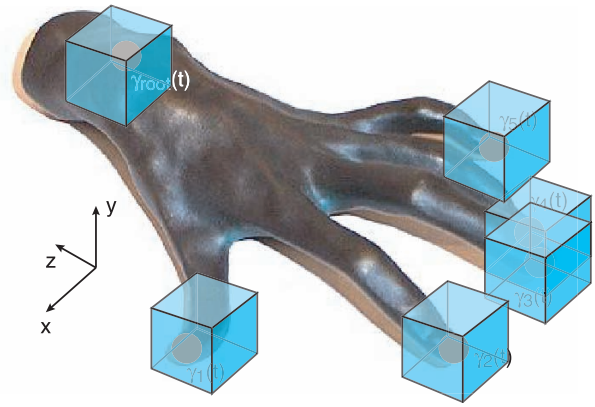

Figure 3.12: Hand model with annotated boundary boxes for hand root and fingers.

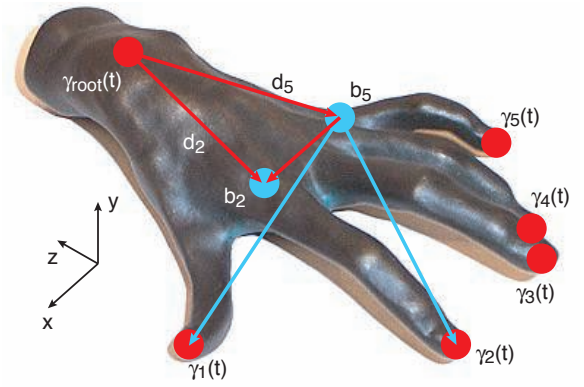

Figure 3.13: Hand model with annotated base positions b_2 and b_5.

Then, for each finger $i = 2, 3, 4, 5$ we require that the triple of vectors

$$\gamma_i^{Space} - b_5,\ b_2 - b_5 = d_2 - d_5,\ \gamma_1^{Space} - b_5$$

has a constant orientation for all t, i.e.:

$$det(\gamma_i^{Space} - b_5, d_2 - d_5, \gamma_1^{Space} - b_5) > 0 \tag{3.30}$$

Dynamic Inequalities

The last set of inequalities deal with a different point of physical reality: dynamics. The movements of the hand root and the finger tips (figure 3.14) have to comply with Newton's

Figure 3.14: Hand model with finger mass for finger 3 and force vector.

law for every $0 \leq t \leq 1$ to reflect human force limitations when acting upon finger masses:

$$m_{root} \cdot \left| \frac{d^2 \gamma_{root}^{Space}}{de(t)^2} \right| < K_{root} \tag{3.31}$$

$$m_i \cdot \left| \frac{d^2 \gamma_i^{Space}}{de(t)^2} \right| < K_i \tag{3.32}$$

Observe that in this first approximation we use point masses m_{root} and m_i, and the constants K_{root} and K_i are upper force limits given by physiological constraints. By applying the chain rule twice, above inequalities can be rewritten as

$$\begin{aligned} &(De(t))^2 \cdot \left| D^2 \gamma_{root}^{Space} \right|^2 + (D^2 e(t))^2 \cdot \left| D\gamma_{root}^{Space} \right|^2 - \\ &2 \cdot De(t) \cdot D^2 e(t) \cdot \left\langle D\gamma_{root}^{Space}, D^2 \gamma_{root}^{Space} \right\rangle \\ &< \left((De(t))^3 \cdot \frac{K_{root}}{m_{root}} \right)^2 \end{aligned} \tag{3.33}$$

and, for $i \in 1, \ldots, 5$

$$\begin{aligned} &(De(t))^2 \cdot \left| D^2 \gamma_i^{Space} \right|^2 + (D^2 e(t))^2 \cdot \left| D\gamma_i^{Space} \right|^2 - \\ &2 \cdot De(t) \cdot D^2 e(t) \cdot \left\langle D\gamma_i^{Space}, D^2 \gamma_i^{Space} \right\rangle \\ &< \left((De(t))^3 \cdot \frac{K_i}{m_i} \right)^2 \end{aligned} \tag{3.34}$$

Summary

Let us shortly recapitulate the previous sections: We are given a number of boundary conditions and polynomial inequalities of the given geometric, anatomic and dynamic constraints. For each segment in the segmented score, the coefficient variables of the polynomial functions for the motion of the hand root and the finger tips need to be calculated in order to satisfy the determined constraints. The resulting cubic spline curves represent constrained motion curves. The next section deals with the specification of the boundary values defined in this section: they need to be correlated to the given hand model, to the keyboard model, and to the segmented score.

3.4.4 Boundary Value Mapping

We have defined a number of constraint equalities in the previous section in symbolic form. In this section we show how the constraints are matched to the keyboard model and to the segmented score to be performed.

First, consider a score that was segmented into n segments according to section 3.4.1. Each segment $i = 1 \ldots n$ contains information for all five fingers, and for all fingers involved in playing an event, i.e., the finger's state is *Attack*, *Hold*, or *Release*, we need to map the score parameters to the physical space of the piano keyboard model given in section 3.4.2. As we have seen earlier, pitch will be mapped to the x-co-ordinate, and the velocity will be mapped to the derivative of y-co-ordinate, denoting the velocity of the key being pressed or released. The absolute mapping of the y-co-ordinate is given by the position when a key is pressed, and of the finger's rest position when after releasing the key. Finally, the absolute mapping of the z-co-ordinate is not directly related to a certain segment, but to anatomic constraints and to playing technique, it actually defines *where* the key will be touched.

Generally, we have to distinguish between white and black keys. Therefore we define a Boolean function $map_w(pitch)$ that is *true* when $pitch$ refers to a white key, and *false* when $pitch$ refers to a black key:

$$map_w(pitch) = \begin{cases} true & \text{if } (pitch \; mod \; 12) \in \{0, 2, 4, 5, 6, 9, 11\} \\ false & \text{otherwise} \end{cases} \tag{3.35}$$

(assuming that $pitch$ corresponds to MIDI pitch with C3 = 60).

Pitch to X Mapping

For the pitch to x mapping, we define the function $map_x(pitch)$:

$$map_x(pitch) = w_w \cdot \left(7 \cdot octave + \begin{cases} 0.0 & \text{if } key = 0 \\ 0.5 & \text{if } key = 1 \\ 1.0 & \text{if } key = 2 \\ 1.5 & \text{if } key = 3 \\ 2.5 & \text{if } key = 4 \\ 3.0 & \text{if } key = 5 \\ 3.5 & \text{if } key = 6 \\ 4.0 & \text{if } key = 7 \\ 4.5 & \text{if } key = 8 \\ 5.0 & \text{if } key = 9 \\ 5.5 & \text{if } key = 10 \\ 6.0 & \text{if } key = 11 \end{cases} \right) \tag{3.36}$$

where w_w is the width of one white key (23.6 mm), $octave = pitch\ div\ 12 - 5$, and $key = pitch\ mod\ 12$. This mapping reflects the typical layout of a standard piano keyboard.

Pitch to Y Mapping

For the y mapping, we define two functions $map_{yr}(pitch)$ and $map_{yp}(pitch)$, the former for a finger's rest (i.e., the finger is above the keys, ready to move, or to press another key) position, the latter for finger's position when a key is pressed:

$$map_{yr}(pitch) = \begin{cases} h_w + offset_{wyr} & \text{if } map_w(pitch) = true \\ h_b + offset_{byr} & \text{otherwise} \end{cases} \tag{3.37}$$

and

$$map_{yp}(pitch) = \begin{cases} offset_{wyp} & \text{if } map_w(pitch) = true \\ offset_{byp} & \text{otherwise} \end{cases} \tag{3.38}$$

where h_w and h_b are the heights of a white key (15 mm) and of a black key (25 mm), respectively. The constants $offset_{wyr}$, $offset_{byr}$, $offset_{wyp}$, and $offset_{byp}$ reflect absolute offsets in rest and pressed positions.

Pitch to Z Mapping

The mapping from pitch to the z co-ordinate consists of two components: first, we have to consider again whether the key in question is white or black, this yields in a base offset in negative z direction. Second, a constant offset specific for each finger is added in order to reflect the hand's anatomic shape. For instance, the middle finger presses a white key not at the border (near $z = 0$), but closer to where the row of black keys start. In contrast, the thumb typically presses closer to the border of the white keys. This model is open to future extensions, for example one could more precisely define variable mappings to take playing style into account.

The mapping function $map_{z\ i}(pitch)$ is defined for each finger $i = 1 \ldots 5$ as follows:

$$map_{z,i}(pitch) = \begin{cases} -offset_{wz,i} & \text{if } map_w(pitch) = true \\ -d_w + d_b - offset_{bz,i} & \text{otherwise} \end{cases} \tag{3.39}$$

where d_w and d_b are the depths of a white key (120 mm) and of a black key (70 mm). As explained above, the offsets $offset_{wz,i}$ and $offset_{bz,i}$ are individual (positive) values for each finger $i = 1 \ldots 5$ that are added to the base offsets given by the key sizes.

Velocity to Y Mapping

Finally, we need to map MIDI velocity (loudness) to the speed of which the finger moves when it is about to press a key. The mapping maps from the constant velocity at the moment when the finger first touches the key with a constant scaling factor to an according MIDI velocity. Any further interaction, such as the force that reacts back from the key to the finger is not taken into account. One of the main problems is that MIDI velocity is not an physical measure for loudness (e.g., decibel) – it is simply a numeric value (in the standard case $0 \ldots 127$) and it is up to the dedicated MIDI device of how to interpret or map this value to an output signal level, for instance. Here, we define the mapping as a scaling with factor $scale_v$ from MIDI value to key speed:

$$map_{dy}(velocity) = scale_v \cdot velocity. \tag{3.40}$$

Assigning the Mappings to Gesture Curves

After we have defined the mapping functions in the previous sections, they are ready to be assigned to the boundary conditions given in section 3.4.3: the x, y, and z mappings deliver boundary values for $\gamma_{i,0}$ (equation 3.20) and for $\gamma_{i,1}$ (equation 3.21). The velocity mapping is used in the y component of the velocity equations 3.22 and 3.23.

3.4.5 A General Method for Solving the Inequalities

Section 3.4.3 concluded with a number of equalities and inequalities that need to be satisfied for the cubic curve segments given in equation 3.10. All of those curve segments are of type

$$f(t) = f_3 t^3 + f_2 t^2 + f_1 t + f_0. \tag{3.41}$$

Again, consider a segmented score, as in the previous sections. For each segment, we need to find the coefficients f_3, f_2, f_1, and f_0, according to the given segments. In this section, we present a method based on Sturm's theorem for cubic splines (Waerden, 1966, §79), that delivers *symbolic* (i.e., not numeric) solutions for the problem. The main reason for finding symbolic solutions is to have the solutions in parameterised form ready for implementation in a programming language.

Theorem 3.1 (Sturm Theorem) *The number of different real roots of an polynomial equation $P(x) = 0$ with real coefficients over an interval $x \in [x_0, x_1]$, the endpoints of which are not roots ($P(x_0) \neq 0$ and $P(x_1) \neq 0$), is equal to the difference between the numbers of sign changes of the* Sturm chains *formed for the interval ends x_0 and x_1.*

A *Sturm chain* is a series of *Sturm functions* $P_0, \ldots, P_n$ and is constructed as follows: given a polynomial function $P(x)$, write $P_0(x) \equiv P(x)$ and $P_1(x) \equiv P'(x)$ and define the Sturm functions by

$$P_n(x) = -\left(P_{n-2}(x) - P_{n-1}(x) \cdot Q_{n-2}(x)\right) \tag{3.42}$$

using Euclid's division theorem. This yields the following chain of Sturm functions,

$$\begin{aligned} P_0 &= Q_0 P_1 - P_2 \\ P_1 &= Q_1 P_2 - P_3 \\ P_2 &= Q_2 P_3 - P_4 \\ &\vdots \\ P_{n-2} &= Q_{n-2} P_{n-1} - P_n \end{aligned}$$

known as a Sturm chain. The chain is terminated when a constant $P_n(x)$ is obtained.

Applying the Sturm Theorem

Since all of our inequalities are polynomial inequalities of the form $P(x) > 0$ for $x \in [0 \ldots 1]$ we can apply the Sturm theorem in order to guarantee that there is no root for $P(x)$ in the given interval. Therefore we build the Sturm chains for all inequalities and require that the number of sign changes of the chains is equal at both interval ends. This procedure leads to a number of equations that need to be solved for the appearing coefficients in the cubic spline equations.

Unfortunately, constructing the Sturm chains for all inequalities, and in particular for the one in equation 3.34, yields very large algebraic terms, and mathematics packages such as Mathematica were not able to find symbolic solutions to the given problem as a whole. However, the next section shows that the method is applicable in principle by giving a solution for a simpler, one-dimensional case.

3.4.6 Solution of the One-Dimensional Case

In this section we shall give an example of the application of the Sturm theorem to solve inequalities for a simple, one-dimensional case. In this case, we have a curve $G : [0, 1] \to \mathbb{R}^2$ which describes the change of pitch without intermission, to be performed by determined finger. The curve $G(t) = (e_G(t), h_G)$ has two components: the curve e_G, measuring physical time, and the pitch curve h_G, measuring physical pitch, as represented by the horizontal distance between the keys of a keyboard. The frozen curve G, as it is shown in figure 3.15, draws the change from pitch $h_1 = 0$ to pitch $h_2 = 5$

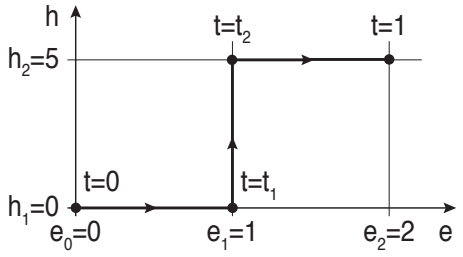

Figure 3.15: "Frozen" gesture curve with changing pitch from $h_1 = 0$ to $h_2 = 5$. Horizontal axis denotes onset time e, vertical axis denotes pitch h.

(think of a fourth leap from C to F), starting at time $e_0 = 0$ for parameter $t = 0$, jumping to h_2 at time $e_1 = 1$ for $t = t_1$, arriving at pitch h_2 at the same physical time e_1 (!) for parameter $t = t_2$, and ending the performance at $e_2 = 2$ for $t = 1$.

The "thawing" deformation is a new curve $g : [0, 1] \to \mathbb{R}^2$ with $g(t) = (e_g, h_g)$, which complies with the Newton inequality

$$m\frac{d^2h}{de^2} < K,$$

where m is the finger's mass, and where K is an upper limit given by the physiological constraints of the performer. The "thawed" curve g is shown in figure 3.16. Evidently, the finger cannot stay fixed on pitch $h_1 = 0$, but has to jump off this position at a physical time $\mu(e_1 - e_0)$, $0 < \mu < 1$, after the start. The main point of the "thawing" calculation is the position, where the jump begins until its ending on pitch $h_2 = 5$. Denote this curve by $\gamma(t) = (e(t), h(t))$ and suppose the curve parameter t ranges from $t = 0$ to $t = 1$, so we have two boundary conditions $h(0) = h_1 = 0$, $h(1) = h_2 = 5$, $e(0) = \mu(e_1 - e_0) = \mu$, and $e(1) = e_1 = 1$. Then the Newton inequality becomes

$$\frac{d^2h}{dt^2} \cdot \frac{de}{dt} - \frac{dh}{dt} \cdot \frac{d^2e}{dt^2} < \left(\frac{de}{dt}\right)^3 \frac{K}{m}, \tag{3.43}$$

which means that this inequality must hold for all $t \in [0, 1]$. As already done previously, we model our curves $e(t), h(t)$ by cubic polynomials:

$$\begin{aligned} e(t) &= e_3t^3 + e_2t^2 + e_1t + e_0 \\ h(t) &= h_3t^3 + h_2t^2 + h_1t + h_0, \end{aligned}$$

where the boundary conditions were given above. After a normalisation of m, K to yield

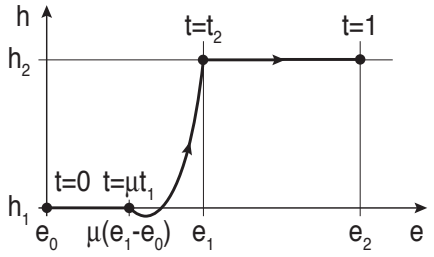

Figure 3.16: "Thawed" gesture curve with changing pitch from $h_1 = 0$ to $h_2 = 5$. Horizontal axis denotes onset time e, vertical axis denotes pitch h.

$K/m = 1$, inequality 3.43 reads as follows:

$$\begin{aligned} P(t) = &- 2h_2e_1 + e_1^3 + 2e_2h_1 + \\ &(-6h_3e_1 + 6e_2e_1^2 + 6e_3h_1)t + \\ &(-6h_3e_2 + 6e_3h_2 + 12e_2^2e_1 + 9e_2e_1^2)t^2 + \\ &(8e_2^3 + 36e_3e_2e_1)t^3 + \\ &(36e_3e_2^2 + 27e_3^2e_1)t^4 + \\ &54e_3^2e_2t^5 + 27e_3^3t^6 > 0. \end{aligned}$$

The Sturm theorem guarantees that no zero of polynomial $P(x)$ occurs in the interval $[0, 1]$ if $P(0), P(1) > 0$, and if the associated Sturm chains $(P(0), P'(0), \ldots)$, $(P(1), P'(1), \ldots)$ have the same number of signature changes. Recall that the Sturm chains are successive Euclidean algorithms, starting with the division with remainder by the derivative P', i.e., $P(t) = A(t)P'(t) + B(t)$.

The Sturm criterion amounts to the fulfilment of a number of polynomial inequalities $S_i > 0, i = 1, \ldots N$, where the polynomials S_i are functions of the curve coefficients $e_3, \ldots e_0, h_3, \ldots h_0$ and the "jumping" coefficient μ, the latter being present from the boundary conditions. One solution to our problem is found by common algorithms (e.g., using Mathematica) and yields $e(t) = 5/8 + t/8 + t^2/4$, $h(t) = -2t + 7t^2$ with $\mu = 5/8$. Figure 3.17 shows a calculated plot of the curve, meaning that the finger first lowers its pitch position and then leaps to the target pitch.

3.4.7 Separating Geometric and Physical Constraints

The previous sections have shown that solving the complete set of constraints using the Sturm theorem imposes major practical problems. As mentioned earlier, an observation on playing technique allows for a new approach in this situation: during rehearsal, a pianist starts playing sequences of notes (for instance fast scales) slowly until the finger movements are correct and internalised. As soon as this stage is reached, he or she can

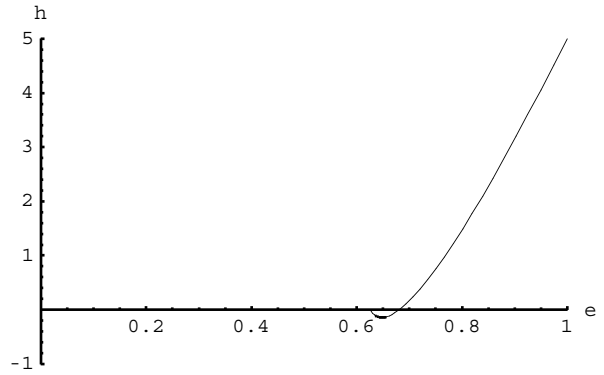

Figure 3.17: Computed plot of "thawed" curve from figure 3.16. Horizontal axis denotes onset time e, vertical axis denotes pitch h.

begin playing the sequence at increasing speeds until the required speed is required. During this process, the shape of the performed gestures is kept as far as possible since drastic changes to the shape would interfere with the internalised movements. Thus, we may perform geometric shaping first, and the proceed with physical shaping without affecting the first step.

Figure 3.18 shows the shaped, physical gesture curve for our example score after geometric shaping has been applied. Figure 3.19 shows the final curve after physical shaping has been performed. The dashed circles show locations where the anticipated leaps from the precedent keys take place.

3.5 Freezing Gesture Curves

While we have just dealt with the construction of symbolic gesture curves, which was denoted by the "thawing" operation in figure 3.4, let us add a remark on the reverse process, the "freezing" of symbolic gesture curves. Since the symbolic gesture spaces are similar to the "Note on", "Note off", and "Velocity" concepts offered by MIDI, the "freezing" operation in the symbolic domain is easy when compared to the construction of a gesture curve: It is basically the transformation of a MIDI file or a real-time MIDI input, respectively, to an event space, for instance defined by E, H, L, and D (onset time, pitch, loudness, and duration).

Important are possible applications of the "freezing" operation: they provide mechanisms for *recording gestures* from given performances. The most simple, but also the most wide-spread use of such a mechanism is MIDI recording, resulting in a recorded score. Further, the "freezing" operation provides a powerful mapping mechanism from a gestural performance device space (e.g. a gesture tracker attached to a computer) to a musical performance space (e.g. a synthesiser).

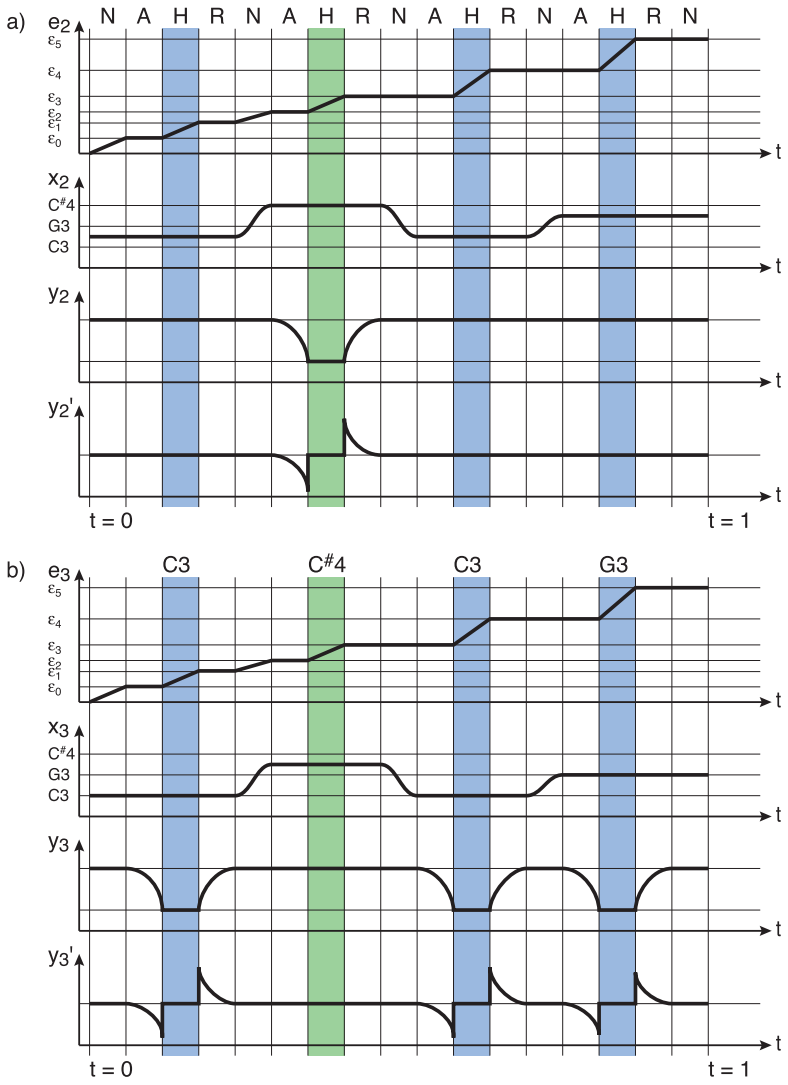

Figure 3.18: Physical gesture curve for fingers 2 (a) and 3 (b), after application of geometric constraints. Curve parameter t is running from 0 to 1 on the horizontal axis.

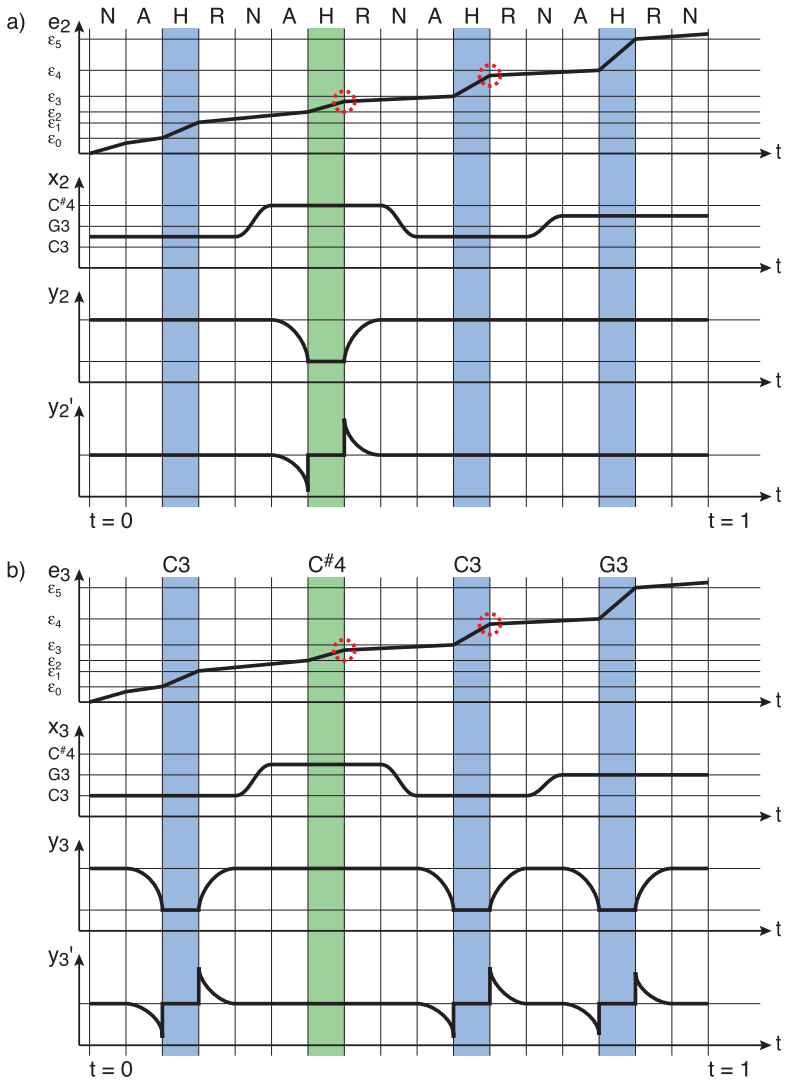

Figure 3.19: Physical gesture curve for fingers 2 (a) and 3 (b), after application of physical constraints. Curve parameter t is running from 0 to 1 on the horizontal axis.

Chapter 4

Implementation

The major part of the theory in chapter 3 has been implemented in a software module called *PerformanceRubette*, which is by itself integrated into the Distributed Rubato Platform (section 2.7). This chapter first gives an overview of the architecture of Distributed Rubato. The overview will be kept short, for a full account refer to (Müller, 2002; Göller and Milmeister, 2003). Then, the implementation details of several independent software components which are used by the PerformanceRubette are given. The PerformanceRubette itself is structured into several modules for computer-aided music performance. The general structure comes from its ancestor, the PerformanceRubette of Classic Rubato. We will show how the Rubette has been seamlessly extended with our concepts of musical gestures. Finally, we will present implementation details which deal with the "freezing" and "thawing" operations and the constraint-based shaping of gesture curves.

4.1 The Distributed Rubato Architecture

Distributed Rubato is the most current version of the Rubato music research platform. While Classic Rubato, its original version, written in Objective C, has been a stand-alone application, Distributed Rubato has been completely rewritten in Java. As the name implies, Distributed Rubato is a *Distributed System*, the software can be run on multiple machines that communicate with each other. Each machine can act both as a client and as a server, therefore Distributed Rubato is well suited for establishing a *Peer-to-Peer* (P2P) network. At the same time, users can still use Distributed Rubato as a single peer application.

The actual functionality of Distributed Rubato is found in the *Rubettes*: those are independent software components that implement and perform arbitrary problems and applications. Communication is established through a well-defined interface based on Forms and Denotators (section 2.4). The Rubette itself does not need to care about communication details or explicit user interface issues, all that is needed is to implement the Rubette interface provided by Distributed Rubato. An example for a typical Rubette would be the

MetroRubette, which performs metric analysis of a musical score: the Rubette provides a Denotator containing descriptive information about itself, takes a score Denotator as input, and delivers metric information as output.

The core of Distributed Rubato is constituted by a number of classes which are required for control and communication. These classes are part of the *Distributed Rubato System Classes* (DRSC). In addition there are packages and classes which are of general use for most Rubettes (such as the matrix package, as we shall see later on). The collection of these classes is called the *Distributed Rubato Foundation Classes* (DRFC).

Figure 4.1 gives an overview of the Distributed Rubato architecture. From the viewpoint of a Rubette (or, a Rubette implementer, respectively), the Foundation Classes are more important than the System Classes, because they provide shared functionality in a very generic manner. The biggest package inside the Foundation Classes is the mathematics package which contains a number of sub-packages:

Module calculus. Provides classes that support and implement algebraic modules and module elements. These classes are essential for the implementation of the Form and Denotator theory (chapter 2.4).

Arithmetics. The arithmetics package contains classes for general arithmetic operations and number theory. Observe that the Java Platform already provides a number of classes for such operations, but for instance lacks support for rationales, or arithmetic strings. Such (missing) features are implemented in the arithmetics package.

Matrices and linear algebra. This package provides classes for matrix operations. In section 4.2.1 we shall give a detailed description of the matrix package.

Parametric curves. This package has been developed to support the concepts of gesture curves given in chapter 3. At the same time, they provide very general and convenient mechanisms for dealing with arbitrary parametric curves, so they are useful for many other applications as well. Section 4.2.2 will present the package in detail.

Yoneda classes. The Yoneda package provides all classes and mechanisms for Forms and Denotators. Since Denotators are used as the primary communication mechanism between Rubettes, they are essential for the behaviour of the whole system.

In addition, the Foundation Classes contain a package for *logic-geometric operations* (LoGeo). These operations allow simple access to the inner structures and data of Denotators. They further provide mechanisms to modify Denotators based on predicates. Finally, the utility package contains a number of classes that do not fit in any of the previous packages. At typical class in the utility package is the `MidiIO` class, providing functionality for reading and writing MIDI files.

4.2 Supporting Components

This section gives an detailed description of two packages which have been essential for the realisation of the PerformanceRubette.

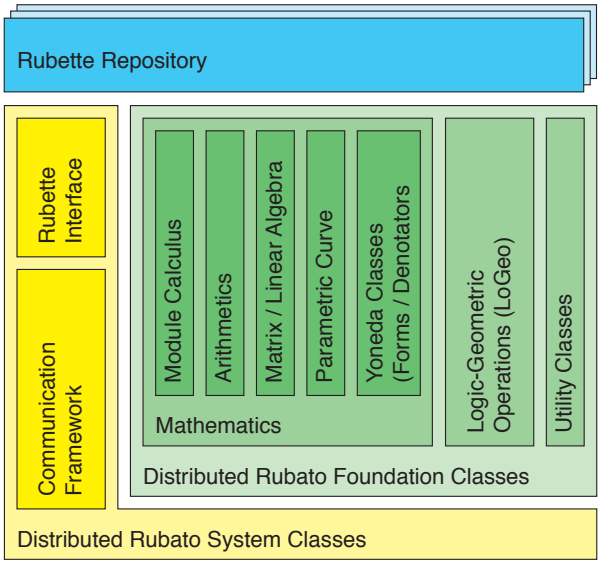

Figure 4.1: Block diagram of the Distributed Rubato architecture.

4.2.1 The Matrix Package

In order to implement matrix and linear algebra operations, we have implemented a rather comprehensive matrix package (refer to figure 4.2 for the inheritance hierarchy). The most notable difference to other Java matrix packages is the support for arbitrary coefficient types. The only requirement on coefficient types is that they are inherited from the interface `Coefficient`, and need to satisfy algebraic ring properties, such as to provide a zero element, a unit element, addition, and multiplication.

The interface `Matrix` by itself is inherited from `Coefficient`, the package therefore implicitly supports block matrices. `Matrix` declares the most common operations on matrices, they can roughly be grouped into the following entities:

Coefficient access. Support for the most basic operations, such as obtaining or setting a coefficient at a given row and column position. For faster numeric operations, direct conversions from and to floating-point values are supported.

Sub-matrix access. Several methods for accessing (either getting or setting) a sub-matrix of the whole matrix are provided.

Matrix layout. This group of methods contains functionality for the inquiry or modification of the matrix' structure. Row or columns can be removed, added, or appended.

Matrix operations. Support for a number of matrix operations. Among others, addition, multiplication, calculation of the rank, the determinant, and the inverse matrix, are found in this group.

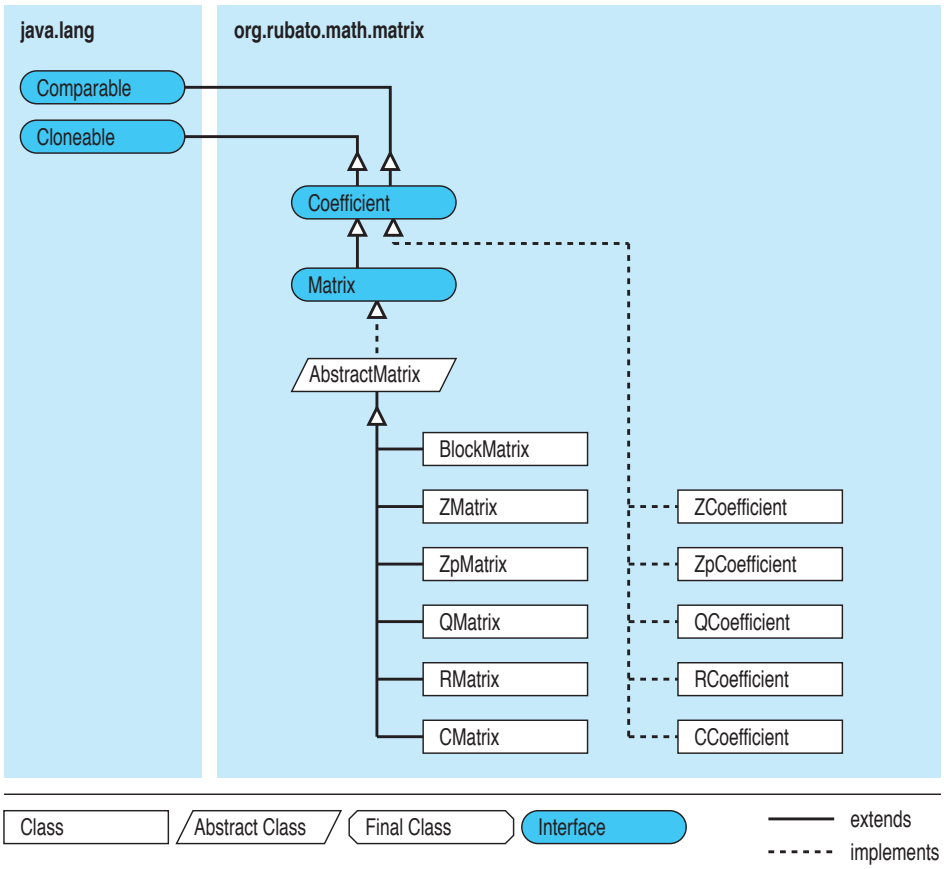

Figure 4.2: Matrix package inheritance hierarchy (the inheritance of classes from `java.lang.Object` is omitted).

Block matrix operations. For block matrices, which can be of arbitrary depth, a number of special operations are required. This typically includes inquiry about the hierarchical structure of the block matrix, and flattening, i.e., conversion from a block matrix to a conventional matrix.

The package defines a number of classes for specific coefficient types: `ZMatrix` contains coefficients of type $\mathbb{Z}$ (i.e., integers), `ZpMatrix` contains coefficients of type $\mathbb{Z}\ mod\ p$. The class `QMatrix` supports coefficients which are of rational type. Finally, `RMatrix` accepts floating-point values as coefficient type.

Since the defined classes in the matrix package are intended to be inherited from, a special mechanism, making use of Java's reflection interface, guarantees that the generated types are as close as possible to their arguments with respect to the inheritance tree. For instance, if a class `MidiEvent` would be inherited from `RMatrix`, then the addition

of two objects of type `MidiEvent` would result in another object of that type. In contrast, the result would be of type `RMatrix` as soon as one of the arguments was not of type `MidiEvent`. Further help comes from `AbstractMatrix` which provides a generic implementation for most matrix operations based on a few basic properties. This makes it easy to inherit from one of above classes without having to rewrite everything from scratch. Of course, sometimes optimised versions are required, this can easily be achieved by overriding the methods provided by `AbstractMatrix` or one of its ancestors.

4.2.2 The Parameteric Curve Classes

The parametric curve package directly reflects the concepts of gesture curves presented in chapter 3: an object of type `Curve` (refer to figure 4.3 for the inheritance and aggregation hierarchy of the curve package) is parameterised in an interval $[t_0, t_1]$ and contains a list of named co-ordinate axes (class `Axis`). Each axis is built of a number of non-overlapping intervals (interface `Interval`), and the sequence of those intervals defines the shape of the curve. There can be arbitrary interval types, the only requirement is that they implement `Interval`, which contains just a few methods to be implemented:

```
public interface Interval extends Comparable {
  public double get(double t);
  public double getT0();
  public double getT1();
  public double getV0();
  public double getV1();
  public double getDV0();
  public double getDV1();
  public double setV0();
  public double setV1();
  public double setDV0();
  public double setDV1();
}
```

□

The most important method is `get(double t)` which returns the interval's value for given parameter t, which must be between t_0 and t_1. The other methods to be implemented are for accessing the border values $v_0 = v(t_0)$ and $v_1 = v(t_1)$, and its derivatives $dv_0 = dv/dt|_{t_0}$ and $dv_1 = dv/dt|_{t_1}$. Observe that the interval's range can only be initialised during construction and then remains immutable to assure consistency of the interval sequence inside a given axis. The curve package comes with a number of existing interval types:

`ConstantInterval` can be used as a constant segment, i.e., $\gamma_{const}(t) = c$ for all $t \in [t_0, t_1]$. Observe that this class does not support setting the derivatives dv_0 and dv_1.

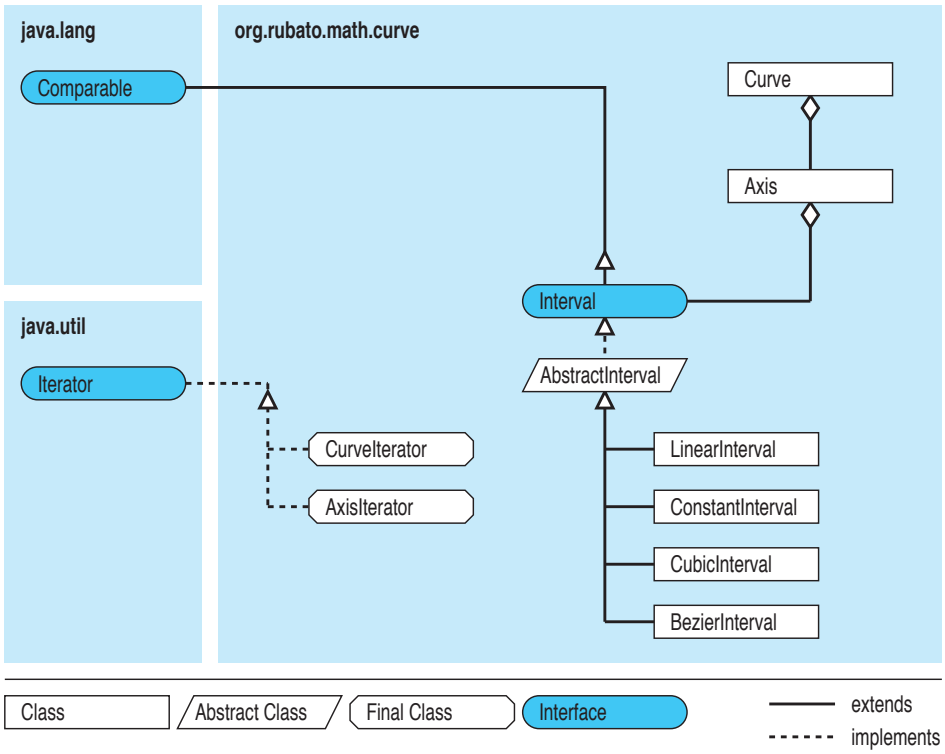

Figure 4.3: Curve package inheritance and aggregation hierarchies (the inheritance of classes from `java.lang.Object` is omitted).

`LinearInterval` is used to construct linear segments, with $\gamma_{linear}(t) = at + b$, where

$$\begin{aligned} a &= \frac{v_1 - v_0}{t_1 - t_0} \\ b &= \frac{v_0 t_1 - v_1 t_0}{t_1 - t_0} \end{aligned}$$

Again, the class does not support setting the derivatives dv_0 and dv_1.

`CubicInterval` allows the definition of a cubic spline $\gamma_{cubic}(t) = at^3 + bt^2 + ct + d$, given by its border values v_0 and v_1, and the first derivatives dv_0 and dv_1. The parameters a, b, c, and d are given as:

$$
\begin{aligned}
a &= \frac{dv_1(t_0^2 - 2t_0t_1 + t_1^2) + dv_0(t_0^2 - 2t_0t_1 + t_1^2)}{t_0^4 - 4t_0^3t_1 + 6t_0^2t_1^2 - 4t_0t_1^3 - t_1^4} + \\
&\quad \frac{-2t_0v_0 + 2t_1v_0 + 2t_0v_1 - 2t_1v_1}{t_0^4 - 4t_0^3t_1 + 6t_0^2t_1^2 - 4t_0t_1^3 - t_1^4} \\
b &= \frac{-dv_1(2t_0^3 - 3t_0^2t_1 + t_1^3) - dv_0(t_0^3 - 3t_0t_1^2 + 2t_1^3)}{t_0^4 - 4t_0^3t_1 + 6t_0^2t_1^2 - 4t_0t_1^3 - t_1^4} + \\
&\quad \frac{3t_0^2v_0 - 3t_1^2v_0 - 3t_0^2v_1 + 3t_1^2v_1}{t_0^4 - 4t_0^3t_1 + 6t_0^2t_1^2 - 4t_0t_1^3 - t_1^4} \\
c &= \frac{-dv_1t_0(-t_0^3 + 3t_0t_1^2 - 2t_1^3) - dv_0t_1(-2t_0^3 + 3t_0^2t_1 - t_1^3)}{t_0^4 - 4t_0^3t_1 + 6t_0^2t_1^2 - 4t_0t_1^3 - t_1^4} + \\
&\quad \frac{6t_0t_1(t_0v_0 - t_1v_0 - t_0v_1 + t_1v_1)}{t_0^4 - 4t_0^3t_1 + 6t_0^2t_1^2 - 4t_0t_1^3 - t_1^4} \\
d &= \frac{-(dv_1t_0^2t_1 + dv_0t_0t_1^2)(t_0^2 - 2t_0t_1 + t_1^2)}{t_0^4 - 4t_0^3t_1 + 6t_0^2t_1^2 - 4t_0t_1^3 - t_1^4} + \\
&\quad \frac{3t_0^2t_1^2v_0 - 4t_0t_1^3v_0 + t_1^4v_0 + t_0^4v_1 - 4t_0^3t_1v_1 + 3t_0^2t_1^2v_1}{t_0^4 - 4t_0^3t_1 + 6t_0^2t_1^2 - 4t_0t_1^3 - t_1^4}
\end{aligned}
$$

`BezierInterval` is equivalent to `CubicInterval` but allows the use of two control points $c_0 = dv_0/3 + v_0$ and $c_1 = v1 - dv_1/3$ instead of derivatives.

Interpolated Curve Access

While it is possible to access all axes and their segments individually through public methods of the `Curve` and `Axis` classes, it is in most cases desirable to have a more mathematical oriented access to curve objects, i.e. in most cases we would like to obtain a vector containing the values for each axis at a given curve parameter t. Assume that we have a instance `curve` of `Curve`, containing n axes. The first way to access the curve values is supported by the method `Curve.get(double t)`. For instance, we can write

```
// obtain curve data at t = 0.5
double t = 0.5
double[] v = curve.get(t);

// do something with v...
```

which would return a vector of dimension n, where n is the number of axes in the curve object.

The main problem of this method is the computational cost when repeatedly calling `get()`, because the interpolation mechanisms hidden in the `Curve` class require an unavoidable amount of recalculation every time the method is called.

In such cases it is common habit in software engineering to adapt the concept of *iterators*, which provide a mechanism for iteration through sequentially organised data. The Java platform provides iterators through the *Java Collections Framework*. The curve package implements the iterator concept in terms of the class `CurveIterator`, which implements `java.util.Iterator`. The following example shows how curve data can be accessed using iterators (again assuming that there is an object `curve` containing valid data):

```
// split the curve interval into 5000 samples
double samples = 5000.0;

// obtain iterator for interval [0, 0.5] at given resolution
java.util.Iterator i = curve.iterator(0.0, 0.5, samples);

// loop from iterator start till end
while(i.hasNext()) {
  // obtain value at current iterator position
  // and increment iterator
  double[] v= (double[])i.next();

  // do something with v...
}
```

□

This code extract would be equivalent to subsequently calling `curve.get()`, with the difference of being a lot faster because the pre-calculated interpolation values are stored inside the iterator. Another advantage is that the obtained iterators can directly be used to transfer curve data into other Java Collections containers, such as vectors, or linked lists.

4.3 The PerformanceRubette

The PerformanceRubette is the central component where a large part of the theory of musical gestures, presented in chapter 3, has been implemented. A major design goal was to integrate the theory into the existing theory of music performance. This theory had already been implemented in the PerformanceRubette of Classic Rubato, but the implementation lacks of two recent developments in performance theory. The first is the capability to deal with arbitrary instrument parameter spaces, and the second is the missing support for musical gestures. Below, we will give a short overview of the overall design of the PerformanceRubette and then focus on how the support for complex instrument spaces and musical gestures has been integrated.

Currently, the PerformanceRubette mainly serves as a testbed for thawing and freezing of gesture curves. While the classes for music performance are present and build up

the essential skeleton of a fully functional new PerformanceRubette, the thorough implementation (and mainly porting work from the former Rubette) is not completed yet as it is beyond the scope of this work, which focused on musical gestures.

4.3.1 Overall Design

The overall design of the PerformanceRubette strictly follows the concepts presented in (Mazzola, 2002c, part VIII) and in particular in (Mazzola, 2002c, chapter 35). Figure 4.4 shows the resulting inheritance and aggregation hierarchies of the package.

From the operational viewpoint, the PerformanceRubette is fed with score data and performance parameters in the Denotator format through the Rubette interface. The Rubette itself does not need to care about the origin of the Denotators, typically they will be provided by Distributed Rubato's GUI (Graphical User Interface), called the *Primavista Browser* (Göller, 2004), which is a Rubette by itself. The class `PerformanceRubette` takes care of the communication to other Distributed Rubato components to the "outside", and performs construction and control of internal classes (as explained later) on the "inside". Further it takes care of conversion of Denotators to the internal data formats. For example, a musical score can be of very high complexity when being present in Denotator format (Montiel Hernandez, 1999). The Denotator is then converted to an internal format, in this case a list of events, which can be used by the other PerformanceRubette classes in an efficient way.

At the core of the performance structure an object of type `Performer` (refer to figure 4.5 (a) for a list of public methods of the `Performer` class) an associated `Instrument` is found. The instrument has its dedicated implementation (with respect to the instrument to be performed). In the case of figure 4.4, this situation is reflected by the realised classes `PianoPerformer` and its associated instrument `PianoInstrument`.

Initially, the instantiated `Performer` object keeps a reference to its instrument, which can be obtained by calling `getInstrument()`, and to the score to be processed, which is accessible through `getScore()`. Then subsequent calls to `applyOperator()`, with a specified operator and a list of weights, yield the hierarchical Stemma structure, which has its root as an object reference to a performance score (class `PS`). The stemma can be accessed by calling `getStemma()`. When the application of operators is finished, the method `perform()` actually initiates the performance calculation. Finally, the resulting performance score can be obtained by calling the method `joinPerformance()`, which merges all calculated performance kernels together.

4.3.2 Support for Complex Instrument Spaces and Musical Gestures

While the functionality in the previous section basically reflects the functionality of Classic Rubato's PerformanceRubette, the new PerformanceRubette has been extended to take the developments of this work into account. The first extension is the support for complex instrument spaces. The original Rubette supported the basic sound parameters E (onset time), H (pitch), L (loudness), D (duration), G (glissando), and C (crescendo).

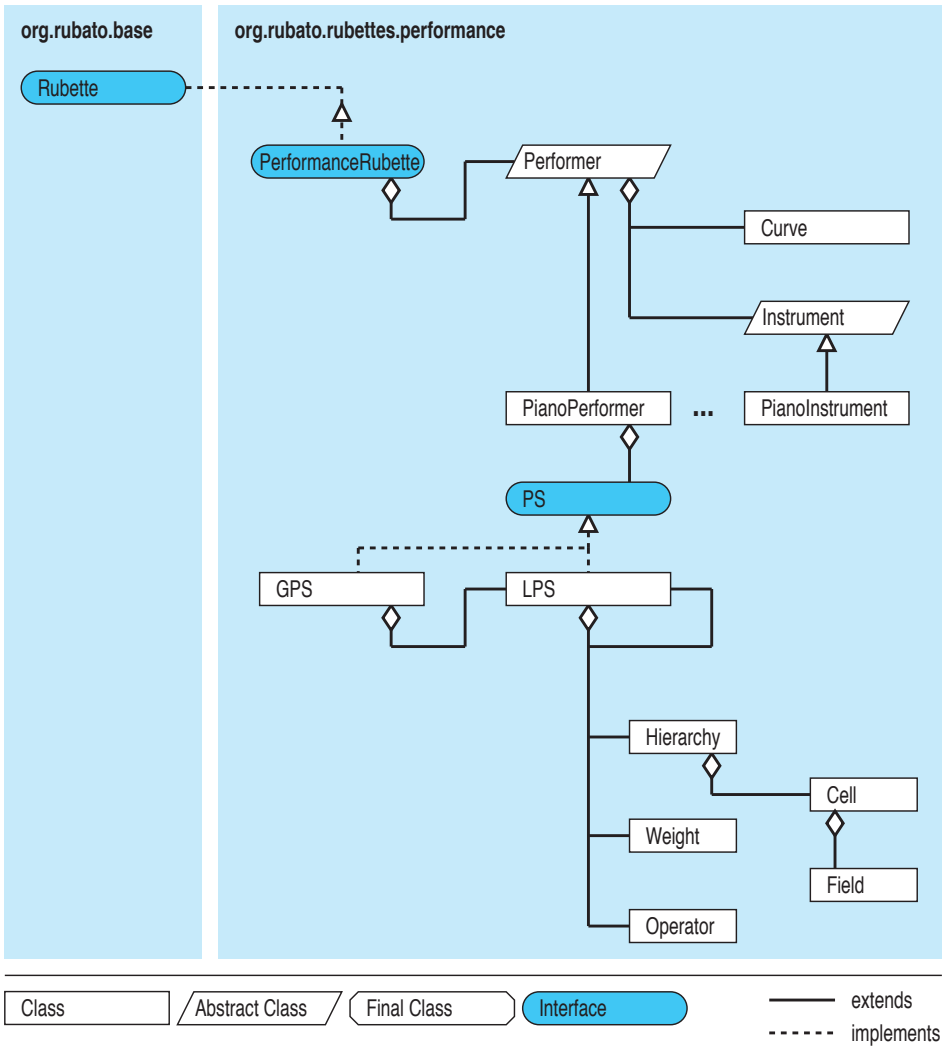

Figure 4.4: PerformanceRubette package inheritance and aggregation hierarchies (the inheritance of classes from `java.lang.Object` is omitted).

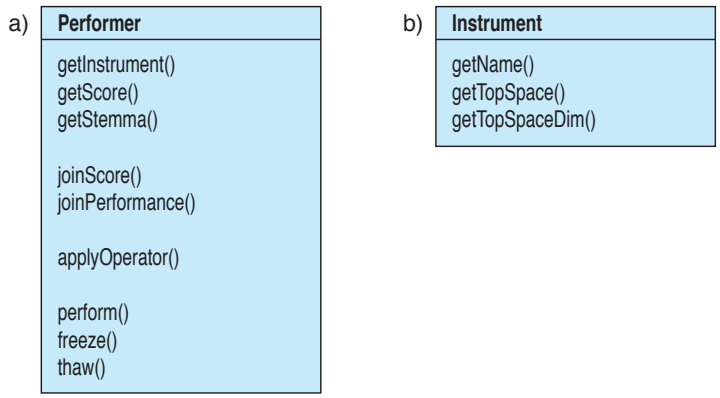

Figure 4.5: Public methods of the `Performer` (a) and `Instrument` (b) classes.

This instrument space has been adequate for many applications, and in particular when the resulting performance was used to generate a MIDI file to be sent to a synthesiser.

However, with the ability to directly control a physical modelling instrument, a fixed parameter set is in many cases not sufficient. One may think of violin performance, where additional sound parameters, e.g., vibrato, or plucked notes, are essential. In order to support this capability, the concept of an abstract performer – instrument pair (as already mentioned above) has been introduced: the concrete implementations of specific instrument types (with their associated instrument spaces) can fully be controlled by inheritance from the `Performer` and `Instrument` classes. Basic properties of the `Instrument` class (4.5 (b)) are its name (accessible through `getName()`), and its space. The method `getTopSpace()` returns a named vector of the instrument space.

In addition two new methods for the support of musical gestures have been added to the `Performer` class: `freeze()` takes as input a symbolic or physical gesture curve, "freezes" it, and returns the corresponding code. On the other hand `thaw()` "thaws" a symbolic or a performance score (including the shaping of the curve), and returns the symbolic or physical gesture curve. The two methods directly reflect the concepts presented in section 3.1.2.

4.4 Efficient Calculation and Shaping of Gesture Curves

This sections describes the core functionality that has been implemented for "thawing" gesture curves from given scores or performances. The procedure is the same for the symbolic and for the physical domain, except that in the physical domain an additional processing step performs the task of shaping the curve according to given physical constraints.

Central to the calculation of gesture curves is the `Performer.thaw()` method, which controls the necessary processing steps and provides them with the given param-

eters. It takes a score, or a performance score, as argument and utilises a number of member variables that are part of the `Performer` class and its derived classes (e.g., the `PianoPerformer` class). The pseudo-code for the method looks as follows:

```
// used member variables:
// Performer performer (this)  the current performer object
// Instrument instrument       the performer's instrument

public Curve thaw(Score score) {
  // first, segment score
  Segment[] segments = segmentScore(score);

  // initialise and prepare curve
  Curve curve = initialiseCurve();

  // construct symbolic gesture curve
  // based on segmented score, performer, and instrument
  constructCurve(curve, segments, this, instrument);

  // if score is a performance score,
  // shape curve based on score, performer, and instrument
  if (score.isPerformanceScore()) {
    shapeCurveGeom(curve, segments, this, instrument);
    shapeCurvePhys(curve, segments, this, instrument);
  }

  // return the resulting gesture curve
  return curve;
}
```

□

The following sections will depict the individual processing steps in detail. The pseudo code extracts follow Java coding style, but with simplified constructs for list processing. Further observe that, for simplicity, the presented data structures are specific to piano performance. The actual implementation is more generic.

4.4.1 Score Segmentation

We have shown in sections 3.3 and 3.4.1 that the segmentation of the score is central to the initial construction and to the shaping of gesture curves. It splits a score into small parts from where elementary gestures (i.e., the individual curve segments) are created. From the viewpoint of implementing the theory, the segmentation of a score has another significance: the organisation of a score into a data-structure that is well suited for the construction of elementary gesture curves and the navigation between them. For example, this includes problems like "what is the next note that finger x will have to play?", or

"what other fingers are involved in playing at a certain onset time E (or e, in the case of a performance score)?"

A segmented score consists of a sorted array of objects of type `Segment`, in ascending order with respect to onset time. Each segment is of a certain type (None, Attack, Hold, and Release, as given in 3.4.1, and contains a reference to the previous and to the next segment in the array. It further contains an array of `Finger` objects, one for each of the ten fingers. Each of these objects contains the type (again, None, Attack, Hold, and Release) specific to the finger, since an individual finger's type may differ from the segment's master type. It contains a reference to the current event (or `null`, if none), and references to the previous and next event that the finger has played, or will have to play, respectively. Finally, each finger object contains `Curve` intervals for E, X, Y, and Z. The following pseudo code extract summarises the two data-structures:

```
public static class Segment {
  // define segment types as constants
  public static final int HOLD = 0;
  public static final int ATTACK = 1;
  public static final int NONE = 2;
  public static final int RELEASE = 3;
  public int type;          // master segment type
  public double startTime;  // start of segment
  public double endTime;    // end of segment
  public Segment prev;      // previous segment
  public Segment next;      // next segment
  public Finger[] fingers;  // array of finger structures
}

public static class Finger {
  public int type;          // segment type for that finger
  public Event event;       // reference to score event
  public Event prev;        // previous score event
  public Event next;        // next score event
  public Interval e;        // curve interval for onset time
  public Interval x;        // curve interval for x axis
  public Interval y;        // curve interval for y axis
  public Interval z;        // curve interval for z axis
}

public static class Event {
  public double e;          // onset time
  public double h;          // pitch
  public double l;          // loudness
  public double d;          // duration
  public double f;          // finger
}
```

□

At this point we may point out that the implemented organisation of a segmented score resembles more the traditional design of algorithms and data-structures than strict object-oriented design. The main reason for this design decision are performance issues: simple data-structures are typically more efficient for algorithmic processing than large and nested class hierarchies.

The actual method, `segmentScore()`, builds the sorted list of segments in three passes: first, it processes the Score and creates Attack and Release segments for all appearing events. The second pass processes the list of created segments and adds None and Hold segments for the remaining placeholders. The final pass completes the finger structures that have only in part been initialised by the first two passes.

```
public Segment[] segmentScore(Score score) {
  Segment[] segments;

  // pass 1: create Attack and Release segments
  foreach (Event event in score) {
    Segment s;

    // create attack segment if it does not exist
    // for the event's onset time
    if (s = segments.find(event.e) != null) {
      s = new Segment(ATTACK, event.e, event.e);
      segments.append(s);
    }

    // add the corresponding finger to the segment
    s.fingers[e.f] = new Finger(ATTACK, event);

    // create release segment if it does not exist
    // for the event's onset time + duration
    if (s = segments.find(event.e+event.d) != null) {
      s = new Segment(RELEASE, event.e + event.d,
                               event.e + event.d);
      segments.append(s);
    }

    // add the corresponding finger to the segment
    s.fingers[e.f] = new Finger(RELEASE, event);
  }

  // pass 2: add None and Hold segments
  foreach (Segment segment in segments) {
    Segment s;

    if (segment.type == ATTACK) {
      s = new Segment(HOLD, segment.endTime);
```

```
      segments.insertAfter(segment, s);
    } else if (segment.type == RELEASE) {
      s = new Segment(NONE, segment.startTime);
      segments.insertAfter(segment, s);
    }
  }

  // pass 3: complete missing data
  foreach (Segment segment in segments) {
    // complete prev and next segment
    segment.prev = segments.prev(segment);
    segment.next = segments.next(segment);

    // complete finger information
    // the method buildFingers completes
    // all finger structures with missing information
    // (e.g. prev and next event, and type)
    segment.buildFingers();
  }

  // finally return segment list
  return segments;
}
```

□

4.4.2 Curve Setup

The task of setting up the curve that will be returned by `thaw()` basically consists of initialising an object of type `Curve` (section 4.2.2), and adding the required co-ordinate axes for the hand root and all fingers. The resulting curve contains 48 axes, 24 for each hand: e, x, y, and z for the hand root; and e, x, y, and z for each finger. Observe, that in contrast to the theory (section 3.4.1), the hand roots and the fingers consist of separate onset axes (e). With the current implementation, these axes will all be identical after the calculation, but with an eye on future developments, we can think of having individual onset axes for the hand roots and the fingers.

4.4.3 Symbolic Gesture Curve Construction

After the score segmentation and the curve setup have been completed, we are ready to construct the symbolic gesture curves for each segment. At this point, the boundary conditions (section 3.4.3), and the boundary value mappings (3.4.4) need to be taken into account. They are implemented as small mapping methods, which will be listed below, but let us first give the pseudo-code for the curve construction method. For simplicity the handling of special cases at the beginning and at the end of the segment lists is omitted.

```
public void constructCurve(Curve curve, Segment[] segments,
                           Performer performer,
                           Instrument instrument) {
  // first, calculate the range for each segment
  // (which is 1 / number of segments)
  double range = 1 / segments.length;

  // iterate through all segments
  for (int i = 0; i < segments.length, ++i) {
    Segment segment = segments[i];

    // calculate t0 and t1 for current segment
    double t0 = range * i;
    double t1 = range * (i + 1);

    // iterate through each finger of current segment
    for (int j = 0; j < segment.fingers.length; ++j) {
      Finger finger = segment.fingers[j];
      double x0, x1, y0, y1, dy, z0, z1;

      // construct curve interval depending on the
      // finger's segment type
      switch (finger.type) {
      case NONE:
        // x: move from previous to next event
        // y: move from previous to next event
        // z: move from previous to next event
        x0 = performer.pitch2x(finger.prev);
        x1 = performer.pitch2x(finger.next);
        y0 = performer.pitch2yr(finger.prev);
        y1 = performer.pitch2yr(finger.next);
        z0 = performer.pitch2x(finger.prev);
        z1 = performer.pitch2x(finger.next);

        // create curve intervals
        f.x = new CubicInterval(t0, t1, x0, x1, 0, 0);
        f.y = new CubicInterval(t0, t1, y0, y1, 0, 0);
        f.z = new CubicInterval(t0, t1, z0, z1, 0, 0);
      break;

      case ATTACK:
        // x: constant at current event
        // y: press key
        // z: constant at current event
        x0 = performer.pitch2x(finger.event);
        y0 = performer.pitch2yr(finger.event);
        y1 = performer.pitch2yp(finger.event);
```

```
  dy = performer.velocity2dy(finger.event);
  z0 = performer.pitch2x(finger.event);

  // create curve intervals
  f.x = new ConstantInterval(t0, t1, x0);
  f.y = new CubicInterval(t0, t1, y0, y1, 0, dy);
  f.z = new ConstantInterval(t0, t1, z0);
break;

case HOLD:
  // x: constant at current event
  // y: constant at pressed key position
  // z: constant at current event
  x0 = performer.pitch2x(finger.event);
  y0 = performer.pitch2yp(finger.event);
  z0 = performer.pitch2x(finger.event);

  // create curve intervals
  f.x = new ConstantInterval(t0, t1, x0);
  f.y = new ConstantInterval(t0, t1, y0);
  f.z = new ConstantInterval(t0, t1, z0);
break;

case RELEASE:
  // x: constant at current event
  // y: release key
  // z: constant at current event
  x0 = performer.pitch2x(finger.event);
  y0 = performer.pitch2yp(finger.event);
  y1 = performer.pitch2yr(finger.event);
  dy = performer.velocity2dy(finger.event);
  z0 = performer.pitch2x(finger.event);

  // create curve intervals
  f.x = new ConstantInterval(t0, t1, x0);
  f.y = new CubicInterval(t0, t1, y0, y1, -dy, 0);
  f.z = new ConstantInterval(t0, t1, z0);
break;
}

// add onset time interval
finger.e = new LinearInterval(t0, t1,
                              segment.startTime,
                              segment.endTime);

// add intervals to curve
curve.addInterval(j + "e", finger.e);
```

```
      curve.addInterval(j + "x", finger.x);
      curve.addInterval(j + "y", finger.y);
      curve.addInterval(j + "z", finger.z);
    }
  }
}
```

□

As above code extract shows, the `Performer` object contains the mapping functions for MIDI values to performer- and instrument-specific co-ordinate values: `pitch2x()` converts from MIDI pitch to the x co-ordinate (corresponds to equation 3.36), `pitch2z()` is the equivalent method for the z co-ordinate (equation 3.39). The methods `pitch2yr()` and `pitch2yp()` map from given pitch to the finger's rest or pressed position (equations 3.37 and 3.38). Finally, the method `velocity2dy()` maps from MIDI velocity to the derivative of the curve when the key is pressed (equation 3.40).

4.4.4 Shaping of Physical Gesture Curves

In the case of the construction of gesture curves in the physical domain, the last step is to shape the calculated curve according to given physical constraints, such that the resulting curve can for example be used to animate a virtual hand model. The current implementation follows the theory and results of section 3.4. Due to the complexity of solving *all* constraints using the Sturm theorem, it only implements a part of it. Since the processing structure is open to extensions, a future implementation incorporating complete constraints calculations could easily be integrated.

As shown in the theory, the shaping occurs in two steps: first, geometric shaping is applied, and then physical shaping is applied as the final step.

Geometric Shaping

Central to the geometric shaping process is the method to find the targets for the next segment, as explained in section 3.4.7. The method `findTargets()` determines, which fingers are involved in playing in the current segment, and where they are supposed to move to the positions required for the upcoming segments.

```
public void shapeCurveGeom(Curve curve, Segment[] segments,
                           Performer performer,
                           Instrument instrument) {
  // first, find targets for segment 0
  double[] prevTargets = findTargets(segments[0]);

  // then, iterate through each segment
  for (int i = 0; i < segments.length; ++i) {
    // find next targets
    double[] nextTargets = findTargets(segments[i]);
```

```
    // iterate through all fingers
    for (int j = 0; j < segments[i].fingers.length; ++j) {
      Finger finger = segments[i].fingers[j];

      // only shape NONE segments, all others are fixed
      if (finger.type == NONE) {
        f.x.setV0(prevTarget[j]);
        f.x.setV1(nextTarget[j]);
      }
    }
  }
}
```

□

After geometric shaping is performed, the curve is in a geometric consistent state. Still remaining is the shaping of the onset time axis, i.e., the physical shaping since the finger movements for Attack and Release segments still occur at infinite velocity in the current state.

Physical Shaping

The remaining step of shaping the time line is based on the results of the one-dimensional solution delivered by the Sturm theorem (section 3.4.6). Shaping occurs in two situations: first, the movements for Attack and Release segments will require the appropriate amount of time. Second, we need to shape the curve in the cases where there is not enough spare time to move a finger from one key to the next. This means that a pressed key has to be released earlier than expected in those cases. The shaping method will also determine the cases, where physical shaping is not possible at all, the main case here is where the required time to shorten an event is larger than the event itself. In that case, the score is not playable with the given constraints.

```
public void shapeCurvePhys(Curve curve, Segment[] segments,
                           Performer performer,
                           Instrument instrument) {
  // each finger is handled individually, so we first
  // iterate through all fingers, then through all segments
  for (int f = 0; f < segments[0].fingers.length; ++f) {
    for (int i = 0; i < segments.length;) {
      // t_avl is the available time for shaping
      // t_req is the required time for shaping
      double t_avl = 0;

      // start at current segment and look ahead until
      // a HOLD segment is reached
      int j = i;
```

```
while (j < segments.length && segments[j].type != HOLD) {
  // required times will be depending on current event
  Finger finger = segments[j].finger[f];

  switch (segments[j].type) {
  case NONE:
    t_avl = finger.e.getV1() - finger.e.getV0();
    t_req = performer.moveTime(finger);
    break;
  case ATTACK:
    t_req += performer.attackTime(finger);
    break;
  case RELEASE:
    t_req += performer.releaseTime(finger);
    break;
  }

  ++j;
}

// advance current segment
// j is the index of the segment to be modified
i = j + 1;
--j;

// get required attack/release times for event at j
double d_attack = performer.attackTime(
                    segments[j].fingers[f]);
double d_release = performer.releaseTime(
                    segments[j].fingers[f]);
if (j == 1) {
  // a) special case at beginning
  segments[j - 1].fingers[f].e.addToV1(-d_attack);
  segments[j - 0].fingers[f].e.addToV0(-d_attack);

} else if (j == segments.length - 1) {
  // b) special case at end
  segments[j - 1].fingers[f].e.addToV1(d_release);
  segments[j - 0].fingers[f].e.addToV0(d_release);

} else if (segments[j - 2].type == ATTACK) {
  // c) ATTACK - HOLD - ATTACK case
  segments[j - 1].fingers[f].e.addToV1(-d_attack);
  segments[j - 0].fingers[f].e.addToV0(-d_attack);

} else if (t_avl >= t_req) {
  // d) enough time for ATTACK and RELEASE
```

```
        segments[j - 2].fingers[f].e.addToV1(d_release);
        segments[j - 1].fingers[f].e.addToV0(d_release);
        segments[j - 1].fingers[f].e.addToV1(-d_attack);
        segments[j - 0].fingers[f].e.addToV0(-d_attack);

      } else {
        // e) not enough time (modification required)
        double t = t_req - t_avl;

        segments[j - 3].fingers[f].e.addToV1(-t);
        segments[j - 2].fingers[f].e.addToV0(-t);
        segments[j - 2].fingers[f].e.addToV1(t + d_release);
        segments[j - 1].fingers[f].e.addToV0(t + d_release);
        segments[j - 1].fingers[f].e.addToV1(-d_attack);
        segments[j - 0].fingers[f].e.addToV0(-d_attack);
      }
    }
  }
}
```

□

In the previous pseudo code extract, the methods `Performer.attackTime()`, `Performer.releaseTime()`, and `Performer.moveTime()` deliver the required times for a certain action depending on the current finger and its associated event.

Chapter 5

Applications

In this chapter, we will present realised applications that make use of the theory and implementation of the previous chapters. We will also shortly highlight what the direction for the future developments of those applications would be. The first application, the animation of body parts, more precisely the hands of virtual music performers, may be seen as the main target application of the theory on gesture curves. The next section will show how gesture curves can be used for the generation and animation of abstract objects; these animations can be used for interactive audio-visual live performances with different sensing inputs.

Another application is the sound synthesis of musical instruments using gestural inputs. Using gesture curves for sound synthesis can help to improve the quality of realism and expression in virtual instruments. Finally, an outlook is given on how gesture curves could be used for music composition.

5.1 Virtual Music Performers

We have noted in chapter 3 that the calculation of gesture curves can be seen as a computational pre-stage for automated animation of human body parts. Thus, for the animation of a virtual performer's hands playing the piano we have used a number of existing components, some of them are used for the calculation of gesture curves, and some of them are fed with the resulting sampled gesture curves. Figure 5.1 shows the "big picture" of how our software, i.e. the PerformanceRubette, has been integrated into the process of animating a human hand model.

The process begins with two data-sets: the first is the musical data to be processed, typically a performance score with supplementary fingering information. The second is a human hand model that provides the required constraints, and that can be animated by a modelling and animation software, as Alias' Maya. For the audio part, the musical data is directly processed by an audio synthesiser, as Apple's QuickTime in our case, resulting in an audio track, ready for being mixed to the video track.

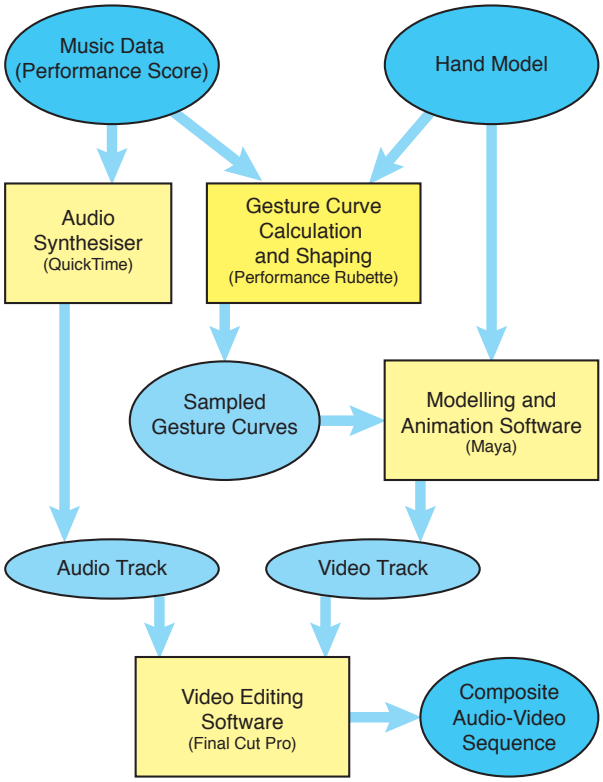

Figure 5.1: The integration of gesture curve calculation into the process of animating a human hand model.

For the animation part, the PerformanceRubette first processes the provided musical data, i.e. it "thaws" the performance score according to chapter 3. The shaping of the generated gesture curves processed together with the given constraints provided by the hand model (the hand model was provided by the Computer Vision Group at ETHZ (Bray et al., 2004)). Then, the shaped gesture curves are sampled and exported into a file. Figure 5.2 shows the resulting gesture curves for the simple score example from section 3.3. For simplicity, the curves for all uninvolved fingers and the hand root has been omitted. The file is read into Maya using a short Mel script (Mel is Maya's integrated scripting language), where for every curve samples corresponding animation keyframes are created.

Since the PerformanceRubette only delivers the positions of the finger tips and the hand root, inverse kinematics (IK) issues are solved by using Maya's internal IK solver. Maya then renders the animation sequence, and the resulting video frames are stored in a video track. Finally, the audio track and the video track are mixed together using a video editing software, in this case Apple's Final Cut Pro. This is a manual step and particular attention has to be kept to audio-video-synchronisation. The resulting sequence can then

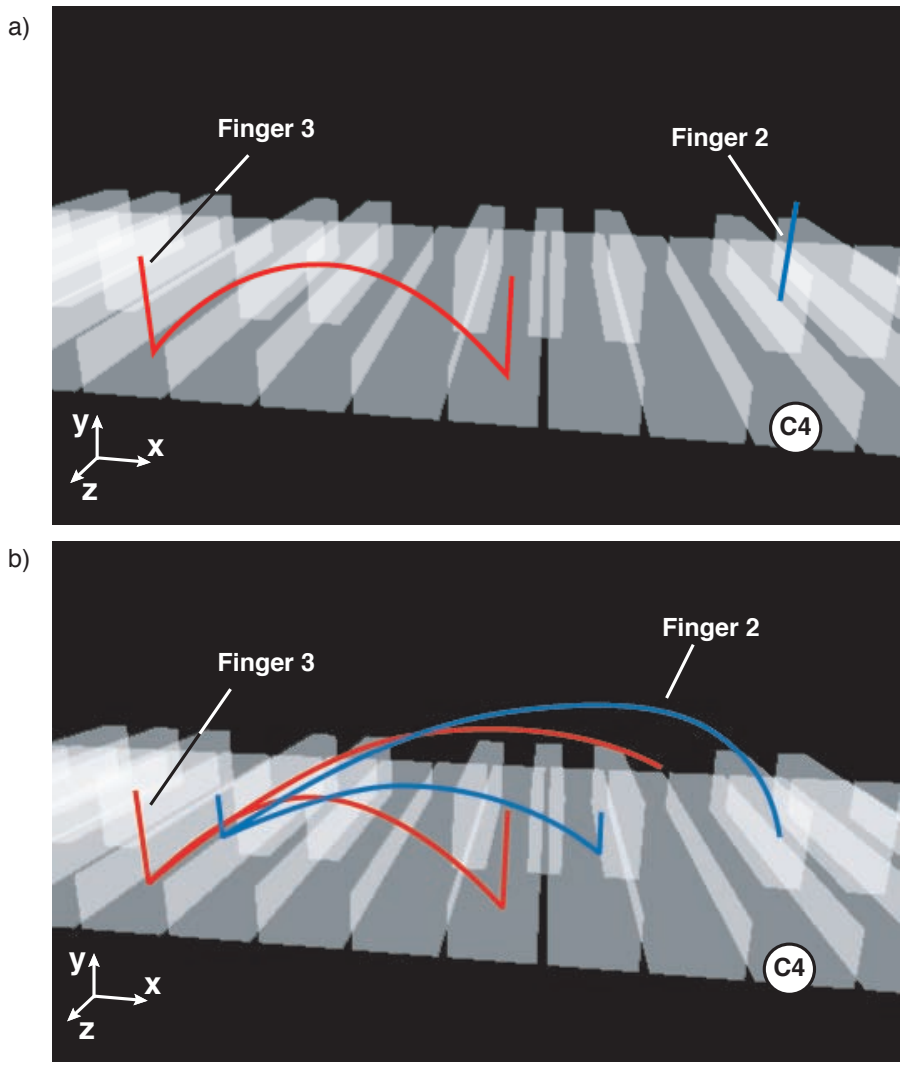

Figure 5.2: Unshaped (a) and shaped (b) gesture curves for the movements of the finger tips of fingers 2 and 3 when performing the example score from section 3.3.

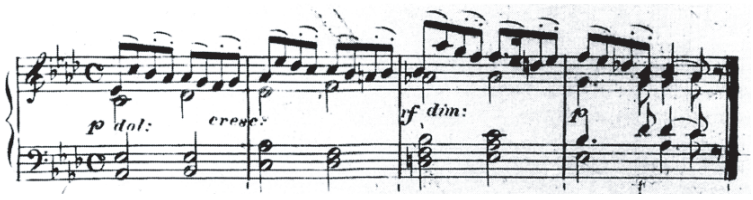

Figure 5.3: Exercise from Carl Czerny's *Vollständige theoretisch-praktische Pianoforte Schule*, Op. 500.

be exported, or it can be used for further processing.

We have applied this processing method to the right hand performance of an exercise from Carl Czerny's *Vollständige theoretisch-praktische Pianoforte Schule*, Op. 500, of which the score is given in figure 5.3. Figure 5.4 shows two frames of the resulting video sequence. The video sequence is found on the accompanying CD-ROM (Appendix A).

While our set-up provides a semi-automatic method for the animation of a performer's hands playing the piano, it is rather inconvenient when used with larger music pieces, or when used in conjunction with animation of other objects because of the manual overhead required for the animation of the keyboard keys. A future version that could be used for production-grade animation of human hands could for example be built as a Maya plug-in that takes a performance score and the hand model data as inputs and then generates the necessary animation curves. In such a case, manual overhead could be avoided to a large part.

5.2 Gesture-controlled Abstract Objects

The previous section dealt with the application of gesture curves to the animation of human body parts. This section shows how gesture curves can be used for the generation and animation of abstract objects. The method has successfully been used in interactive audio-visual live performances with Soundium 2 (section 2.8).

The basic idea is to capture audio data, MIDI data, and data provided by aribtrary sensory devices such as pressure sensors, movement detectors, and 3D motion trackers (figure 5.5). The captured data is then analysed in order to extract features that can be used for the shaping process of gesture curves. For audio data this typically includes beat detection and higher level rhythmic analysis, and the extraction of other audio features, such as levels and sound colours. The extracted features can then be used to generate and manipulate abstract graphical objects, ranging from simple circle, disks, and polygons to complex, fully textured 3D objects.

Since Soundium 2 is an *interactive* performance application, it permits the user to define parameterised gesture curves at run-time without the interruption of the running performance. The curves can be of arbitrary dimension, but practise has shown that 1D

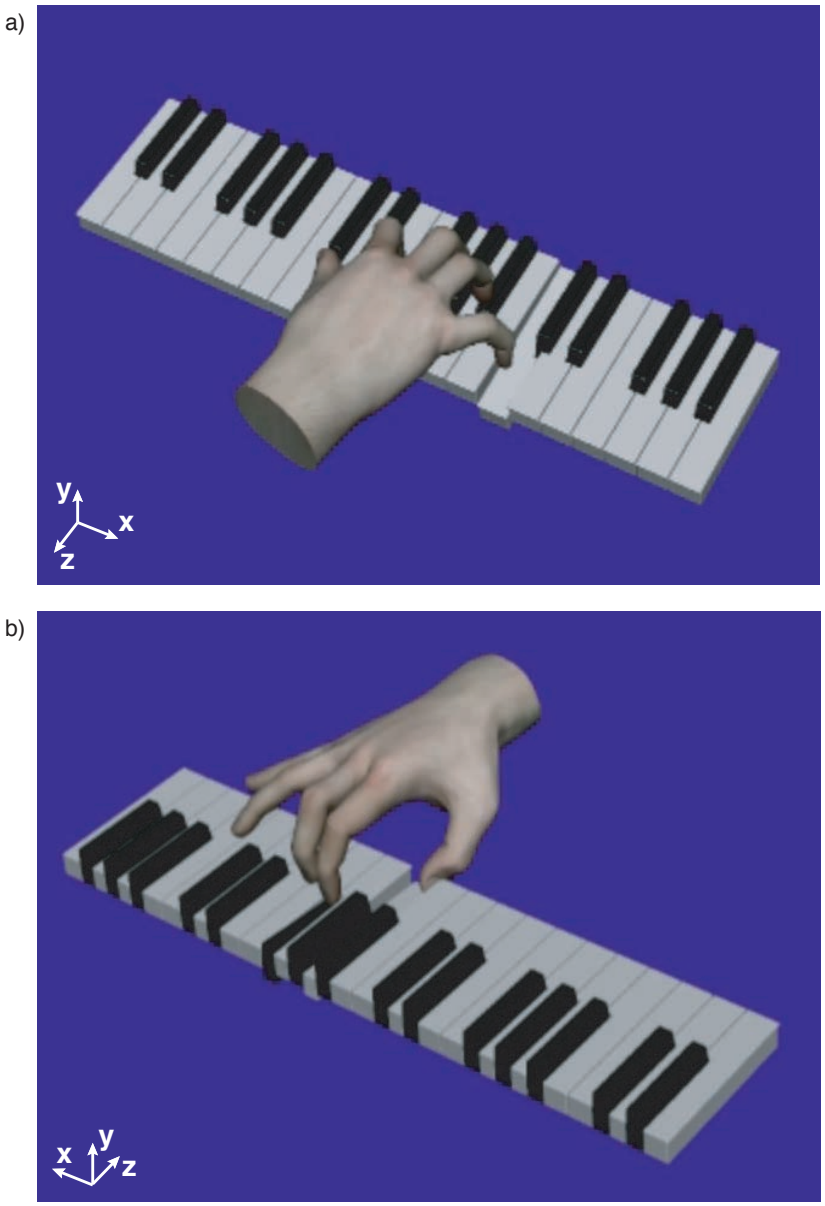

Figure 5.4: Two frames of the resulting video sequence of the Czerny exercise, a) top-rear view, and b) top-front view.

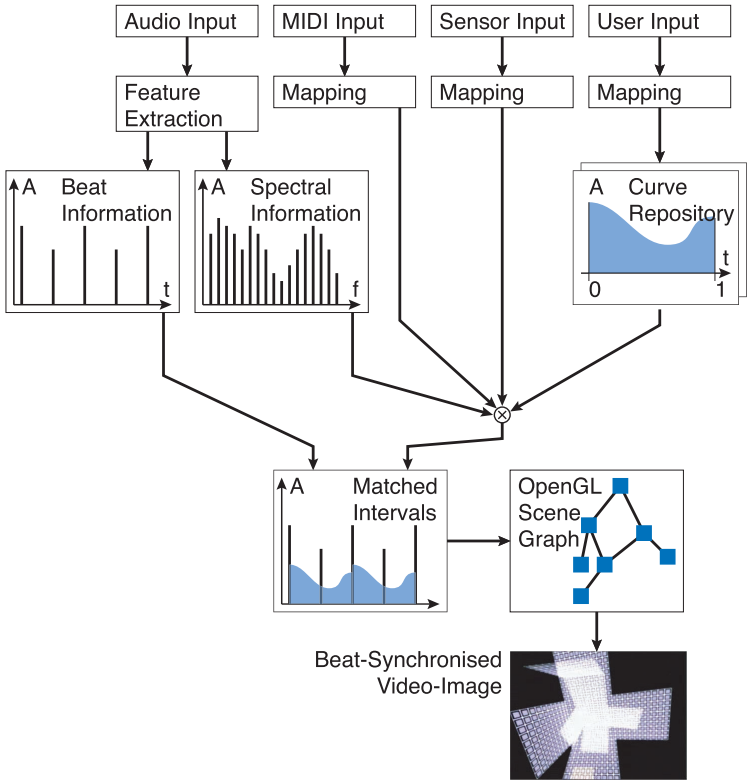

Figure 5.5: Data flow for typical setup in an interactive performance application where gesture curves are used to animate visual objects.

curves (for simple movements of objects, such as a rotation around a fixed axis), 3D curves (for more complex movements of objects, such as a real 3D translation), and 9D curves (for a completely specified manipulation of an object in terms of translation, scale and orientation) are most useful.

The parameter range $[0, 1]$ of a curve is then mapped to a (typically integral) multiple of the extracted beat intervals. The curves by themselves can be manipulated using the gesture curve operations given in section 3.2. The curves can not only be used for the manipulation of visual objects, they can for instance also be used for the manipulation of audio data, therefore providing a method of feedback to the system of audio-visual inputs and outputs.

While the real-time synchronisation of audio and video has been available in live performance tools before, the use of parameterised curves for the generation and manipulation of audio and video objects provides a powerful and unified method to performing artists and drastically enhances the expressive capabilities of a performance tool such as

Soundium 2. In a further step, the concepts could be incorporated into professional video-editing tools, for instance supporting the user with automatic audio-based video cutting. Triggers for scene transitions and parameters for image manipulation, such as brightness, colour correction, etc., could then be mapped to parameterised curves which are synchronised to a number of audio or control streams.

Example video sequences taken during several live performances using Soundium 2 are found on the accompanying CD-ROM (Appendix A).

5.3 Gesture-controlled Sound Synthesis

As pointed out in section 2.5, many modern sound synthesis methods offer large parameter spaces in order to control of the generated sound. However, not all possible values in those parameter spaces produce "good" sounds: for instance a simple frequency modulation (FM) synthesiser is typically controlled by a number of base and carrier frequencies and associated amplitudes. Now, only very specific values for the frequencies and amplitudes produce sounds that can be used. Most other values, and random values in particular, produce sounds with poor spectrums, thus sounding like sine waves, or they produce noise. This observation becomes even more serious for physical modelling synthesis methods. Thus, a desired solution would be a much simpler parameter space that provides an intuitive method of control for the produced sounds and that prohibits the generation of "useless" sounds. Required in this case is a mapping from such a simple parameter space to the parameter space of an individual synthesis method. Gesture curves can provide such a mapping: they allow the control of a virtual instrument through gestural parameters.

In (Laczko, 2003), the physical modelling software Modalys (section 2.5.3) was used to model the mechanics of a piano. The input to the model is a gesture curve describing the movement of a finger tip pressing a key, thus consisting of position and velocity parameters. The input parameters are then evaluated, and the event of pressing or releasing a key is mapped to the synthesis of the piano hammer hitting the string.

While the model for a virtual piano is very simplistic and the gestural control of pressing a keyboard key is rather limited, it demonstrates that gesture curves *can* be used for the control of virtual instruments. Figure 5.6 (a) illustrates the work flow for a typical setup: An input score is "thawed", the resulting curves are then mapped to the synthesiser's parameter space. A conversion step would convert the curve data to the synthesiser's internal data representation, for example, in Laczko's work this is done using a script to create a Modalys input file from given curve data.

In future steps, the method could be applied to instruments with highly complex gestural parameter spaces, such as a virtual violin. In such a case, the gesture curves would represent the complex and manifold ways of playing a violin, including vibrato movements for one hand, or the many different ways of controlling the bow, and other ways to excite the strings, for example plucking.

Another future application is the use of gesture curves in conjunction with electromechanic music instruments, such as electric violins, or electric wind instruments (figure

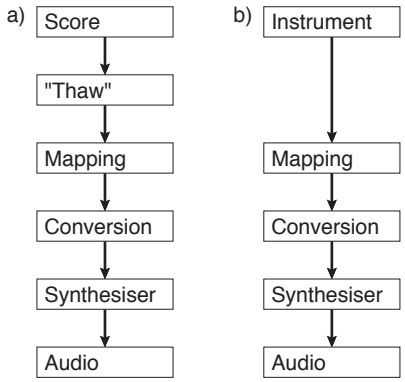

Figure 5.6: Data flow for typical setup when gesture curves are used for sound synthesis: a) when a performance score is "thawed" and the resulting curves are used as input, and b) when a gestural input device is used as input.

5.6 (b)). Most of these instruments map the usually rich gestural input parameter sets to a MIDI stream, thus loosing a large amount of the gestural information provided by the player. Instead of this mapping, the sensory inputs can be used to directly generate gesture curves, which can then be used to map the input parameter space to the parameter space of the appropriate synthesis method, as explained above.

5.4 Composition with Musical Gestures

In (Müller and Mazzola, 2003a), a number of questions were raised how musical gestures, and gesture curves in particular, could be used for music composition. We have pointed out in section 2.1.1 that composers often use a mental image of a musical gesture to compose, so providing appropriate tools to compose directly with a concept of musical gestures would help in realising this mental images in a more direct manner.

One of the main difference between traditional composition methods and a possible gestural composition method is that the former typically deals with discrete music events, such as individual notes. In contrast, the latter would have to take into account the *continuous* nature of gesture curves: music events would not be represented as individual entities, but rather as continuous groups of events, where polyphony for example would implicitly be covered. The concepts of operations on gesture curves, as presented in section 3.2, would provide powerful mechanisms to enhance the expressiveness of the composition process.

For the realisation of a gestural composition software we see two main challenges: the first deals with the syntax and the semantics of gesture curves. One would have to define a language of musical gestures that are useful for composition. In chapters 3 we have mainly dealt with "microgestures" – gestures that represent only very basic movements and do not contain a large amount of *meaning* by themselves. Therefore further research

on how more complex gestures can be built out of simpler ones is required. The second challenge deals with the question of how an appropriate user interface (either graphical, or including specialised devices) for gestural composition would look like. Powerful but intuitive mechanisms for the creation and manipulation of gesture curves would definitely be required.

Chapter 6

Conclusions and Future Work

This thesis has dealt with the reconstruction of musical gestures from given musical material, typically a music performance. The reconstructed gestures are represented by parameterised gesture curves, which can be used to semi-automatically animate hand movements of virtual performers, or to animate abstract objects, which then behave synchronously to the original performance. Gesture curves can further be used to control virtual musical instruments by providing a mapping between gestural parameter spaces and the instrument parameter space, and for gesture-based composition techniques.

The question of reconstructing musical gestures has two origins. The first is the shift that took place in music listening practice when music recording became available: since only the acoustic aspect of a music performance is recorded, all other aspects, and in particular performers' physical behaviour, i.e. their gestures is lost. Further, synthetically produced sounds are often completely unrelated to musical gestures, since the previously strict connection between environment, performer, instrument, and listener is blurred. Thus, the capability to reconstruct musical gestures from a given performance provides a way to make computer-generated music performance more realistic.

The second origin arises from a recent trend in computer animation: at the same time as computer animations are getting more realistic, they are getting more complex as well. Today, the complexity has reached a level where automatic or semi-automatic tools supporting the animators are mandatory. Our work contributes to this trend by providing a mechanism for the automatic generation of gestural movements in animations based on a data source – in our case the animation of a human hand playing an instrument according to a music performance.

In this concluding chapter, we will present the results achieved, and discuss existing problem areas. An outlook will hi-light possible future directions of our work.

6.1 Results

We have presented a novel mathematical framework which allows for the representation and manipulation of musical gestures. Gestures are represented by parameterised gesture curves. By defining a number of mathematical operations, such as addition, scaling, or concatenation, the gesture curves can be manipulated, and complex curves can be built out of simple ones. The framework fits into the existing model of musical performance of mathematical music theory, where a music performance is defined as a transformation from a symbolic score space to a physical performance space. The gesture curves are defined in raised pairs of spaces, the symbolic and the physical gesture space, and, analogous to the performance transformation $\wp_{Score}$ there is a gesture transformation $\wp_{Gesture}$ from the symbolic to the physical gesture space.

The observation that music notation originated from Neumes and became increasingly abstract over the centuries supports our thesis that notes in a musical score can be seen as an abstract notation of musical gestures, or, in other words, that notes are "frozen" gestures. Thus, the vertical relationships, i.e. between the score space and the symbolic gesture space, and between the performance space and the physical gesture space, are defined by a "freezing" operation, which transforms the gesture space to the score space, and a "thawing" operation, which transforms in the opposite direction. We have shown how this vertical operations can be realised, more precisely how gesture curves can be constructed from a given score. At the core of these operations is the segmentation of the musical score into small intervals, where the parameterised curves are placed.

While the generated curves are valid in the symbolic domain, they need further manipulation in the physical domain to account for given anatomic, geometric and physical constraints. We have defined a constrained hand model that is well-suited for piano performance. The manipulation of the originally constructed curves is realised in terms of a shaping, in order to satisfy the constraints given by the hand model. The constraints, formulated as inequalities, can be solved using Sturm's theorem for cubic splines, and we have presented the solution for the one-dimensional case.

The major part of the theory has been implemented in Java in a software module for computer-aided musical performance, the PerformanceRubette. The software has successfully been used to calculate gesture curves for musical scores, which then have been feed into the animation software Maya, where they were used to animate a hand model playing on a keyboard. The software is further used for audio-visual live performances, where gesture curves control the generation and manipulation of abstract visual objects. Finally, we have shown how gesture curves can control physically modelled virtual instruments.

6.2 Problem Areas

The main problem we encountered deals with the complexity of the human hand. Defining a complete set of constraints for possible finger movements results in a large number of inequalities, some of which are high-order polynoms. The attempt to symbolically solve

the whole set of constraints using Sturm's theorem at once imposes major problems even to advanced mathematics packages such as Mathematica. While the solution for the one-dimensional case is relatively simple, the resulting terms are getting immensely large and complex when adding additional constraints and we were unable to obtain a solution for the whole set. We have attacked the problem by splitting geometric and physical constraints, which opens the path for a solution. Future work would include the realisation of the new method and an evaluation of its correctness.

Another problem arises from the method of using an external hand model and the integration of the different software components in terms of a loosely coupled network. The hand model we used comes with its own constraints, and they need to be mapped to the constraints defined in section 3.4.3. Currently, this mapping was done manually, a step that has to be repeated every time a change is made to the hand model. As mentioned in section 5.1, the situation could be improved with the development of an integrated software module as a plug-in for Maya. The module would then contain the hand model, whose constraints are tightly coupled to the gesture curve calculation and shaping mechanism and to Maya's internal inverse kinematics solver.

6.3 Future Work

From the viewpoint of the presented theory, future work would deal with concepts of how higher level gestures can be represented. While gesture curves are capable of representing complex gestures, particularly in conjunction with the defined operations on gesture curves, the challenging question is how to represent the *meaning* they are carrying. An important step would be the attempt to define a gesture grammar, or a gesture language with the corresponding rules of how these gestures can be used in a meaningful way. While there is almost no existing work on this subject in the field of music research, one could certainly learn from *Labanotation*, an abstract gestural notation for dance choreography, and from the extensive work that has been carried out on sign languages.

In terms of the implementation, the next steps would be to realise the construction of gesture curves of other instruments than the piano, in particular wind and string instruments. While the theory on gesture curves is rather generic, every instrument has its own characteristics, and its own method of gestural control. Thus, the constrained shaping process would have to be refined for each instrument. Ideally, common constraints for different instruments could be applied in a unified manner, which would reduce the amount of work every time a new instrument is realised. Future work would also consider not only the performer's hands, but also arm movements, and ideally the whole body, including facial expression. With those improvements, the task of automatic generation of musical gestures could be realised in a commercial product for computer animation and music educational tools.

We further believe that the development of a gestural composition software (section 5.4) is very promising. The availability of such a product would provide radically novel ways of dealing with music objects to composers.

Finally, future work would examine how the presented theory could be applied to

other types of gestures, such as machine or hand writing, sign languages, or arbitrary body movements. Such work could lead to an universal model of human gestures, and its realisation would have enormous effects in computer animation.

Appendix A

CD-ROM Contents

The following documents are found in the top-level directory of the accompanying CD-ROM:

smg.pdf (PDF) Browseable Acrobat PDF document of this thesis.

czerny.dv (DV-PAL) Animated sequence of right hand performing the exercise from Carl Czerny's *Vollständige theoertisch-praktische Pianoforte Schule*, Op. 500.

czerny.avi (DivX 5.0) Animated sequence of right hand performing the exercise from Carl Czerny's *Vollständige theoertisch-praktische Pianoforte Schule*, Op. 500.

scheinwerfer.mp4 (MPEG-4) Demo Reel and selected video sequences from interactive live performances produced by the Soundium 2 multimedia system. The animations are generated using parametric gesture curves which are matched to the beat of an analysed audio input stream. (Copyright © 2003 Corebounce Association.)

scheinwerfer.avi (DivX 5.0) Demo Reel and selected video sequences from interactive live performances produced by the Soundium 2 multimedia system. The animations are generated using parametric gesture curves which are matched to the beat of an analysed audio input stream. (Copyright © 2003 Corebounce Association.)

rubato Top-level directory of the full Java source code of the Distributed Rubato Framework. Note that this code represents a snapshot as of March 2004. More recent releases of Distributed Rubato will eventually be available online.

The movie files are also available online at:
http://www.ifi.unizh.ch/staff/mueller/diss/

Bibliography

Th. W. Adorno. Der getreue Korrepetitor. In *Gesammelte Schriften*, volume 15. Suhrkamp, Frankfurt am Main, Germany, 1963.

I. Albrecht, J. Haber, and H.-P. Seidel. Construction and animation of anatomically based human hand models. In D. Breen and M. Lin, editors, *Eurographics / SIGGRAPH Symposium on Computer Animation*, 2003.

A. Asperti and G. Longo. *Categories, Types and, Structures. An Introduction to Category Theory for the Working Computer Scientist*. MIT Press, Cambridge, 1991. Currently out of print.

N. Badler, C. Phillips, and B. Webber. *Simulating Humans: Computer Graphics, Animation, and Control*. Oxford University Press, 1993.

N. Badler, M. Palmer, and R. Bindiganavale. Animation control for real-time virtual humans. *Commun ACM*, 42:64–73, 1999.

M. Bray, E. Koller-Meier, P. Müller, L. Van Gool, and N. N. Schraudolph. Hand tracking by rapid stochastic gradient descent using a skinning model. In *Proceeedings of the Conference on Visual Media Production*, London, UK, 2004.

R. Bristow-Johnson. Wavetable synthesis 101, a fundamental perspective. In *AES Preprints*. Audio Engineering Society, Los Angeles, 1996.

B. Buchholz, T. J. Armstrong, and S. A. Goldstein. Anthropometric data for describing the kinematics of the human hand. *Ergonomics*, 35(3):261–273, 1992.

C. Cadoz and M. M. Wanderley. Gesture – Music. In M. M. Wanderley and M. Battier, editors, *Trends in Gestural Control of Music*, pages 71–94. IRCAM Centre Pompidou, 2000.

R. Chase. Examination of the hand and relevant anatomy. *Plastic Surgery*, 7:4247–4284, 1990.

I. Choi. Gestural primitives and the context for computational processing in an interactive performance system. In M. M. Wanderley and M. Battier, editors, *Trends in Gestural Control of Music*, pages 139–171. IRCAM Centre Pompidou, 2000.

J. Chowning. The synthesis of complex audio spectra by means of frequency modulation. *Journal of the Audio Engineering Society*, 21(7):526–534, 1973.

R. B. Dannenberg. An on-line algorithm for real-time accompaniment. In *Proceedings of the 1984 International Computer Music Conference*, pages 193–198, San Francisco, 1984. International Computer Music Association.

DIN8996. *Klaviatur für Pianos und Flügel*. Beuth Verlag, Berlin, Wien, Zürich, 1985.

G. ElKoura and K. Singh. Handrix: Animating the human hand. In D. Breen and M. Lin, editors, *Eurographics / SIGGRAPH Symposium on Computer Animation*, 2003.

S. Göller. *Object-Oriented Rendering of Complex Abstract Data*. PhD thesis, University of Zürich, Zürich, Switzerland, 2004.

S. Göller and G. Milmeister. Composition on distributed rubato by affine transformations and deformations of musical structures. In *Proceedings of the 2003 International Computer Music Conference*, San Francisco, 2003. International Computer Music Association.

H. Heijink, P. Desain, H. Honing, and L. Windsor. Make me a match: An evaluation of different approaches to score-performance matching. *Computer Music Journal*, 24(1): 43–56, 2000.

H. Honing. Expresso, a strong and small editor for expression. In *Proceedings of the 1992 International Computer Music Conference*, pages 215–218, San Francisco, 1992. International Computer Music Association.

F. Iazzetta. Meaning in musical gesture. In M. M. Wanderley and M. Battier, editors, *Trends in Gestural Control of Music*, pages 259–268. IRCAM Centre Pompidou, 2000.

J. Kim, F. Cordier, and N. Magnenat-Thalmann. Neural network-based violonist's hand animation. In *Proc. Computer Graphics International (CGI 2000)*, pages 37–41, 2003.

S. Laczko. *Gestik als Medium*. Lizentiatsarbeit. University of Zürich, Zürich, Switzerland, 2003.

J. Langner, R. Kopiez, Ch. Stoffel, and M. Wilz. Real-time analysis of dynamic shaping. In C. et al. Woods, editor, *Proceedings of the Sixth International Conference on Music Perception and Cognition*, Keele, Staffordshire, UK: Department of Psychology, 2000.

S. K. Liddell. *Grammar, Gesture, and Meaning in American Sign Language*. Cambridge University Press, Cambridge, 2003.

S. Matsuda and T. Rai. DIPS: the real-time digital image processing objects for MAX environment. In *Proceedings of the 2000 International Computer Music Conference*, San Francisco, 2000. International Computer Music Association.

G. Mazzola. Inverse performance theory. In *Proceedings of the 1995 International Computer Music Conference*, pages 533–540, San Francisco, 1995. International Computer Music Association.

G. Mazzola. The topos geometry of musical logic. In G. Assayag, H. H. Feichtinger, and J. Rodrigues, editors, *Mathematics and Music*. Springer, Berlin et al., 2002a.

G. Mazzola. Structures mathématiques dans l'interprétation et l'improvisation. In *Séminaire Mamux*, Paris, France, 2002b. IRCAM Centre Pompidou.

G. Mazzola. Semiotic aspects of music. In Posner et al., editor, *Semiotics: A Handbook on the Sign-Theoretic Foundations of Nature and Culture*, volume III, pages 3119–3188. de Gruyter, W., Berlin, Germany, 1999.

G. Mazzola. *The Topos of Music*. Birkhäuser, Basel, Switzerland, 2002c.

G. Mazzola and S. Göller. Performance and interpretation. In G. Johannsen and G. de Poli, editors, *Human Supervision and Control in Engineering and Music*, volume 31(3), pages 221–232. Special Issue. Journal of New Music Research, 2002.

G. Mazzola and S. Müller. Physical shaping of symbolic gesture curves. In *Proceedings of the 2003 International Computer Music Conference*, San Francisco, 2003. International Computer Music Association.

G. Mazzola and O. Zahorka. The RUBATO performance workstation on NEXTSTEP. In *Proceedings of the 1994 International Computer Music Conference*, San Francisco, 1994a. International Computer Music Association.

G. Mazzola and O. Zahorka. Tempo curves revisited: Hierarchies of performance fields. *Computer Music Journal*, 18(1):40–52, 1994b.

J. McCartney. Rethinking the computer language: SuperCollider. *Computer Music Journal*, 26(4):61–68, 2002.

J. McDonald, T. Jorge, K. Alkoby, A. Berthiaume, R. Carter, P. Chomwong, J. Christopher, M. J. Davidson, J. Furst, B. Konie, G. Lancaster, L. Roychoudhuri, E. Sedgwick, N. Tomuro, and R. Wolfe. An improved articulated model of the human hand. *The Visual Computer*, 17:158–166, 2001.

M. Montiel Hernandez. El denotador: Su estructura, constructión y papel en la theoría matemática de la musica. Master's thesis, Universidad Nacional Autónoma de México (UNAM), México DF, 1999.

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

S. Müller. Audio expert system. Master's thesis, Swiss Federal Institute of Technology (ETHZ), Zürich, Switzerland, 1998.

S. Müller. Parametric gesture curves: A model for gestural performance. In G. Mazzola, T. Noll, and T. Weyde, editors, *Proceedings of the 3rd International Seminar on Mathematical Music Theory*. EPOS, Osnabrück, 2003.

S. Müller. Computer-aided musical performance with the Distributed RUBATO environment. In G. Johannsen and G. de Poli, editors, *Human Supervision and Control in Engineering and Music*, volume 31(3), pages 233–238. Special Issue. Journal of New Music Research, 2002.

S. Müller and G. Mazzola. Perspectives of a gestural composition theory. In *Séminaire Mamux*, Paris, France, 2003a. IRCAM Centre Pompidou.

S. Müller and G. Mazzola. The extraction of expressive shaping in performance. *Computer Music Journal*, 27(1):47–58, 2003b.

R. Parncutt. Recording piano fingering in live performance. In B. Enders and N. Knolle, editors, *Proceedings of KlangArt-Kongress*, pages 263–268, Osnabrück, Germany, 1995.

C. Parrish. *The Notation of Medieval Music*. Norton, W. W., New York, 1957.

M. Puckette and C. Lippe. Score following in practice. In *Proceedings of the 1992 International Computer Music Conference*, pages 182–185, San Francisco, 1992. International Computer Music Association.

M. Pukette. MAX at seventeen. *Computer Music Journal*, 26(4):31–43, 2002.

R. Putz and R. Pabst. *Atlas of Human Anatomy – Volume 1: Head, Neck, Upper Limb*. Lippincott Williams & Williams, Philadephia, 13th edition, 2001.

C. Ramstein. *Analyse, Représentation et Traitement du Geste Instrumental*. PhD thesis, Institut National Polytechnique de Grenoble, Grenoble, France, 1991.

T. Richardson and K. R. Wood. *The RFB Protocol*. ORL, Cambridge, 1998.

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

U.-J. Rüetschi. *Denotative Geographical Modelling – An Attempt at Modelling Geographical Information with the Denotator System*. Diploma Thesis. University of Zürich, Zürich, Switzerland, 2001.

S. Schubiger. *Automatic Software Configuration - A Model for Service Provisioning in a Dynamic and Heterogenous Environment*. PhD thesis, University of Fribourg, Fribourg, Switzerland, 2002. No. 1393.

S. Schubiger-Banz and S. Müller. Soundium2 – an interactive multimedia playground. In *Proceedings of the 2003 International Computer Music Conference*, San Francisco, 2003. International Computer Music Association.

J. Stange-Elbe. *Analyse- und Interpretationsaspekte zu J. S. Bachs 'Kunst der Fuge' mit Werkzeugen der Objektorientierten Informationstechnologie. Habilitation.* University of Osnabrück, Osnabrück, Germany, 1999.

J. Sundberg. Music performance research – an overview. In J. Sundberg, L. Nord, and R. Carlson, editors, *Music Language, Speech and Brain*. MacMillan Press, London, UK, 1991.

N. P. M. Todd. The dynamics of dynamics: A model of musical expression. *Journal of the Acousitcal Society of America*, 91(6):3540–3550, 1992.

B. Vercoe. The synthetic performer in the context of live performance. In *Proceedings of the 1984 International Computer Music Conference*, pages 275–278, San Francisco, 1984. International Computer Music Association.

I. Wachsmuth. Communicative rhythm in gesture and speech. In A. Braffort, R. Gherbi, S. Gibet, J. Richardson, and D. Teil, editors, *Gesture-Based Communication in Human-Computer Interaction*, pages 277–290. Stringer-Verlag, Heidelberg, 1999.

B. L. Waerden. *Algebra I*. Springer, Berlin et al., 1966.

Ch. Wagner. The pianist's hand: Anthropometry and biomechanics. *Ergonomics*, 31(1): 97–131, 1988.

J. Weissmann. *Human Computer Interaction with a Command Sign Language*. PhD thesis, University of Zürich, Zürich, Switzerland, 2003.

B. Zagonel. *O Que É Gesto Musical*. Brasiliense, São Paolo, 1992.

Index