Gestural Composition with Arbitrary Musical Objects and Dynamic Transformation Networks

A Dissertation
Submitted to the Faculty of the Graduate School
of the University of Minnesota
by

Florian Thalmann

in Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy

Advised by Guerino Mazzola
Co-Advised by Michael Cherlin

June 2014

UMI Number: 3630273

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.



UMI 3630273

Published by ProQuest LLC (2014). Copyright in the Dissertation held by the Author.

Microform Edition © ProQuest LLC.
All rights reserved. This work is protected against
unauthorized copying under Title 17, United States Code



ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

© Florian Thalmann 2014
All rights reserved

Contents

List of Figures vi

List of Tables xii

I Composition, Theory, and Analysis, and the Dimension of Embodiment 1

1 Introduction: Musical Distances and Lewin's Vision 2

2 Musical Ontology and Conceptualization 14

2.1 Mazzola's Topography of the Ontology of Music 17

2.2 The Dimension of Embodiment 20

2.3 Communication Between the Levels of Embodiment 26

3 The Paradigm Shift towards Gestures in Music Theory and Analysis 29

3.1 Facts and Set Theory 30

3.1.1 Generalized Set Theory 32

3.2 Processes and Transformational Theory 33

3.2.1 Transformation Graphs and Networks 33

3.2.2 Transformations in the Category of Modules 34

3.2.3 Denotators and Forms 36
3.3 Gestures in Music Theory 42
3.3.1 Gesture Theories 43
3.3.2 Gesture Theory as an Extension of Transformational Theory . 44

4 Facts, Processes, and Gestures in Composition and Improvisation 49
4.1 Some Thoughts on Composition and Improvisation 49
4.2 Gestures in Improvised Music 52
4.3 Embodiment and Interactive Composition Systems 54
4.3.1 Thinking and Making Facts or Objects 55
4.3.2 Composition Systems and Processes 60
4.3.3 Gestural Interaction with Composition Systems 62
4.4 Rubato Composer 74
4.4.1 Brief History 74
4.4.2 A Platform for Forms and Denotators 78
4.4.3 Rubettes and Networks 84
4.4.4 Where are the Gestures? 87

II The BigBang Rubette and the Levels of Embodiment 89

5 Introduction to BigBang 90

6 Facts: BigBangObjects and their Visualization and Sonification 93
6.1 Some Earlier Visualizations of Denotators 94
6.1.1 Göller's PrimaVista Browser 94
6.1.2 Milmeister's ScorePlay and Select2D Rubettes 97
6.2 An Early Score-based Version of BigBang 99

6.2.1 The Early BigBang Rubette's View Configurations 102
6.2.2 Navigating Denotators 107
6.2.3 Sonifying Score-based Denotators 107
6.3 BigBangObjects and Visualization of Arbitrary $Mod^{@}$ Denotators . . 109
6.3.1 A Look at Potential Visual Characteristics of Form Types . . 110
6.3.2 From a General View Concept to BigBangObjects 114
6.3.3 New Visual Dimensions 118
6.4 The Sonification of BigBangObjects 119
6.5 Examples of Forms and the Visualization of their Denotators 122
6.5.1 Some Set-Theoretical Structures 122
6.5.2 Tonal and Transformational Theory 125
6.5.3 Synthesizers and Sound Design 128

7 Processes: BigBang's Operation Graph 137

7.1 Temporal BigBangObjects, Object Selection, and Layers 140
7.1.1 Selecting None and Lewin's Transformation Graphs 141
7.1.2 The Temporal Existence of BigBangObjects 142
7.1.3 BigBangLayers 144
7.2 Operations and Transformations in BigBang 146
7.2.1 Non-Transformational Operations 147
7.2.2 Transformations 154
7.3 BigBang's Process View 159
7.3.1 Visualization of Processes 159
7.3.2 Selecting States and Modifying Operations 160
7.3.3 Alternative and Parallel Processes 162
7.3.4 Structurally Modifying the Graph 165

7.3.5 Undo/Redo 166

8 Gestures: Gestural Interaction and Gesturalization 167

8.1 Formalizing: From Gestures to Operations 169

8.1.1 Modes, Gestural Operations, and the Mouse 170

8.1.2 Affine Transformations and Multi-Touch 177

8.1.3 Dynamic Motives, Sound Synthesis, and Leap Motion 180

8.1.4 Recording, Modifying Operations and MIDI Controllers 183

8.2 Gesturalizing and the Real BigBang: Animated Composition History 185

8.2.1 Gesturalizing Transformations 185

8.2.2 Gesturalizing Other Operations 188

8.2.3 Using Gesturalization as a Compositional Tool 189

III Implementation and Examples 191

9 Architecture and Implementation 192

9.1 The Architecture of BigBang 192

9.2 BigBangModel 195

9.2.1 BigBangOperationGraph 198

9.2.2 BigBangDenotatorManager 200

9.2.3 BigBangObjects 204

9.3 BigBangView 205

9.3.1 The View's Model Classes 206

9.3.2 The (Sub)View Classes and their Controller Classes 208

9.4 Synthesizer and MIDI Classes 209

9.5 Other Implementation Details 211

9.5.1 Duplicating 212

9.5.2 Saving and Loading 213
9.5.3 Testing BigBang 215

10 Musical Examples **219**

10.1 Some Example Compositions 219
10.1.1 Transforming an Existing Composition 220
10.1.2 Gesturalizating and Looping with a Simple Graph 222
10.1.3 Drawing UPIC-Like Motives and Transforming 224
10.1.4 Drawing Time-Slices 227
10.1.5 Converting Forms, Tricks for Gesturalizing 229
10.1.6 Gesturalizing a Spectrum 232
10.1.7 Using Wallpapers to Create Rhythmical Structures 234
10.2 Improvisation and Performance with BigBang 234
10.2.1 Improvising by Selecting States and Modifying Transformations 236
10.2.2 Playing Sounds with a MIDI Keyboard and Modifying Them . 238
10.2.3 Playing a MIDI Grand Piano with Leap Motion 238
10.2.4 Playing a MIDI Grand Piano with the Ableton Push 239
10.2.5 Improvising with 12-Tone Rows 243

Bibliography **246**

List of Figures

2.1 Mazzola's three-dimensional topographic ontology. 19

2.2 Mazzola's extension of the ontology cube to a hypercube by introducing the dimension of embodiment. 24

2.3 The three levels of embodiment and the arrows symbolizing communication between them. 27

3.1 The brief *EulerScore* defined above, in staff notation. 40

3.2 A sample gesture with skeleton $\Gamma = (\{0,1,2,3\}, \{(0,0),(0,1),(1,2),(1,2),(2,3)\})$ and space $X = \mathbb{R}^3$. 46

4.1 The types of facts in computer-assisted composition and how they are typically converted into each other. 56

4.2 An example from LilyPond, (a) as code and (b) as staff notation (source: http://lilypond.org/text-input.html). 57

4.3 Ableton's sequencer *Live* with a few audio (sound waves) and MIDI (piano roll) tracks. 58

4.4 Xenakis's *UPIC* system. 59

4.5 A Max/MSP patch I created to test the velocity calibration of a MIDI grand piano. Each visible object performs an action or transformation while the musical objects, MIDI notes, travel along the connective lines. 62

4.6 Wessel and Wright's two-way scheme of gestural interaction between humans and computers. 65
4.7 A juxtaposition of (a) Max Mathews's *Radio Batons*, and (b) *Leap Motion*. 73
4.8 Mazzola demonstrating his $\mathbb{M}(2,\mathbb{Z})\setminus\mathbb{Z}^2$-*O-Scope* to Herbert von Karajan and an audience at the *Salzburger Musikgespräche* in 1984. 75
4.9 The Atari ST software *Presto*. 76
4.10 The *OrnaMagic* component of *Presto*. 77
4.11 The *Rubato* software on NeXTSTEP. 78
4.12 A performance field calculated by Müller's *EspressoRubette*. 79
4.13 The architecture of *Rubato Composer*. 80
4.14 The graphical user interface of *Rubato Composer*. 81
4.15 Creating the example *EulerScore* with *Rubato Composer*'s Denotator Builder 83

5.1 A network including the *BigBang* rubette and its view next to it. . . 91

6.1 The *Di* of Göller's *PrimaVista* browser. 95
6.2 A denotator visualized in *PrimaVista* using Pinocchios (satellites) of varying size and differently positioned extremities (subsatellites). . . . 97
6.3 The *Select2D* rubette showing a *Score* denotator on the $Onset \times Pitch$ plane. 99
6.4 The early *BigBang* rubette showing a *Score* in piano roll notation. . . 104
6.5 The early *BigBang* rubette visualizing a *Score* in a more experimental way. 105
6.6 The early *BigBang* rubette showing a $MacroScore$ with two levels of satellites. 108

6.7 The new *BigBang* rubette visualizing *Pitch* denotator in every visual dimension. 123
6.8 A *PitchSet* simultaneously visualized using several visual characteristics. 124
6.9 A *PitchClassScore* drawn with ascending and descending lines to show the cyclicality of the space. 125
6.10 A *Progression* where pitches adopt the visual characteristics of their anchor chord. 127
6.11 A *GeneralScore* with some *Notes* and *Rests* shown on the *Onset* × *Pitch* plane. 128
6.12 A *Spectrum* shown on *Loudness* × *Pitch*. 130
6.13 A constellation of eight *HarmonicSpectra* with different fundamental *Pitches* and *Overtones*. 131
6.14 An instance of a *DetunableSpectrum*, where the fundamentals of the *Overtones* are slightly detuned. 132
6.15 An *FMSet* containing five carriers all having the same modulator arrangement, but transposed in *Pitch* and *Loudness*. 133
6.16 A composition based on a **Limit** of a *SoundSpectrum* (*Pitches* at *Onset* 0) and a *Score* (*Pitches* with *Onsets*). 135

7.1 A factual notion of a composition above, versus a dynamic notion below. 139
7.2 A *MacroScore* with its second level of satellites being selected. Note that the y-axis is *SatelliteLevel* to facilitate the selection. The x-position and colors of the objects hint at their chaotic arrangement on the *Onset* × *Pitch* plane. 141

7.3 A table illustrating how BigBangObjects keep track of their location. Each column is a state of a simple composition process with an $FMSet$. The rows are what each of the objects save: a path for each of the state the object exists at, pointing to the denotators corresponding to the objects ($FMNodes$) are at, at the respective state. Note that all paths are assigned according to the x-axis here ($Loudness$ in $FMSet$). . . . 144

7.4 An $FMSet$ distributed on three layers, represented by the rectangluar areas at the top. Layer 0 is inactive and inaudible (its $Partials$ in the facts view are greyed out), layer 1 is active and selected (its $Partials$ are darkened), and layer 2 is active, but not selected (normal bright color). 146

7.5 A composition drawn in $Onset \times Pitch$ with a shaped third dimension represented by color. 150

7.6 A two-dimensional wallpaper in early *BigBang*. 152

7.7 A $Score$ alteration in early *BigBang*. (a) shows the unaltered $Score$, whereas in (b) $Pitch$ and $Duration$ are altered with $dg_1 = 0\%$ and $dg_2 = 100\%$. 155

7.8 A small composition made with copy-and-translate (bottom left), copy-and-rotate (top left), copy-and-scale (top right), and copy-and-reflect (bottom right). 156

7.9 A *BigBang* operation graph showing a linear composition process. . . 161

7.10 An operation graph with two alternative processes. 163

7.11 An operation graph with parallel processes. 164

8.1 The shearing *DisplayTool* shown while a copy-and-shear is performed. 171

8.2 The three most common two-dimensional multi-touch gestures: (a) drag, (b) pinch, and (c) twist. 178
8.3 The components resulting from a three-finger gesture. 179
8.4 The two three-finger gestures for (a) shearing and (b) reflection. . . . 180
8.5 An *FMSet* denotator consisting of a carrier and five modulators defined by the fingertips of the user. 182
8.6 A wallpaper with a motif defined by the fingers of a hand. 183

9.1 The general architecture of *BigBang*. Solid arrows refer to direct method calls, while dashed arrows show update procedures. 194
9.2 Some of the most important model classes of *BigBang*. 196
9.3 A simplified view of the process initiated by calling the *translateObjects(...)* method for a transposition T_7. The corresponding *TranslationTransformation*, named *upAFifth* here, is represented at different stages of its existence. 197
9.4 The class hierarchy of *BigBang*'s operations in *org.rubato.rubettes .bigbang.model.operations*. 199
9.5 *BigBang*'s denotator manager and its helper classes in package *org.rubato.rubettes.bigbang.model.denotators*. 204
9.6 Some of the classes in *BigBang*'s view system (package *org.rubato .rubettes.bigbang.view*). 207
9.7 Some of the classes of *BigBang*'s sound and MIDI classes (packages *org.rubato.rubettes.bigbang.view.io* and *org.rubato.rubettes.bigbang.view .player*). 210

10.1 A transformation of Scarlatti's Sonata *K003* resulting in a pulsating bass sound. 221

10.2 A growing and rotating Scarlatti *K002* during gesturalization. 223
10.3 The *Onset* × *Pitch* plane of the *UPIC*-like composition. 225
10.4 The *Onset* × *Pan* plane of the *UPIC*-like composition. 226
10.5 The *Pan* × *Pitch* plane on which drawing took place. 228
10.6 The resulting slices seen on the *Onset* × *Pitch* plane. 228
10.7 The facts view shows the *Texture* at state 1, with rate and duration represented by height and width, respectively. The process view shows the graph generating the entire composition. 230
10.8 The *Spectrum* during gesturalization. For a video, see the link above. 233
10.9 The two wallpaper patterns in this example. 235
10.10 The *Spectrum* and process that form the basis of this brief improvisation. 237
10.11 Two piano hands drawn with Leap Motion on the *Pitch* × *Loudness* plane. 240
10.12 The initial motive and the simple sequential graph. 242
10.13 The sequential graph with the initial state selected, showing the original twelve-tone row. 244

List of Tables

4.1 Some examples of controllers with gestural capabilities. 74

8.1 *BigBang*'s operations and their gestural capabilities. 169

Part I

Composition, Theory, and Analysis, and the Dimension of Embodiment

Chapter 1

Introduction: Musical Distances and Lewin's Vision

Even though music and mathematics shared a long past, as two sister arts with the potential of explaining and illustrating the order of the universe, they had increasingly grown apart since the seventeenth century. A serious interest in seeing the two domains closely connected has reawakened only in the second half of the twentieth century. Today, both in music theory and composition, mathematical models and procedures find an ever widening range of applications. In music theory, certain subdomains such as set theory or neo-riemannian theory are now a crucial part of the university curriculum. In composition, every practitioner is confronted with serial or probabilistic methods at some point in their education. Nevertheless, many scholars and musicians stay at a safe distance from mathematics when it comes to more advanced applications. Satyendra and others suggest that the reason for this may be a certain language barrier created by the ever more complex formalisms developed by the few specialists that actively contribute to the field.[1] However, it is not only the

[1] Ramon Satyendra. "An Informal Introduction to Some Formal Concepts from Lewin's Transformational Theory". In: *Journal of Music Theory* 48 (2004), pp. 99–141, p. 99.

complexity of mathematics that keep musicians away from the field. In recent years, a number of scholars have criticized mathematical methods in theory and composition for their positivistic attitude, their determinism, and for being too distant and abstracted from music, music-making, and listening.[2]

There are several reasons for this perceived distance, which can in fact all be expressed by partial distances that can be perceived between mathematics and both analysis and composition. First, there is a *conceptual distance*, which is essentially what Satyendra claims. With the evolution of modern mathematics, structures and procedures became increasingly complex and may no longer be immediately graspable and understandable by a wide audience. A composer may need to invest a considerable amount of time to understand Xenakis's musical applications of stochastic processes[3] or Hook's uniform triadic transformations.[4] Second, there is a *temporal distance*, both because of the atemporality of mathematical structures and the time taken to understand or apply mathematics. When composers decide to create musical structures using mathematical procedures they typically need months of calculation and notation time until they are able to hand a score to performers. If they make a mistake, or the result is not to their liking, they need even more time, to experiment and modify their musical results. The same is true for mathematical music theory, where, for example, an analyst may ignore expressive features or historical connections within the music in question while taking time to identify set classes and calculate relationships between them. Third, there may be a *generative distance*,

[2]See for instance Richard Taruskin. "Review of Forte, The Harmonic Structure of the Rite of Spring". In: *Current Musicology* 28 (1979), p. 119; George Perle. "Pitch-class set analysis: An evaluation". In: *Journal of Musicology* (1990), pp. 151–72. For an overview of the debate see Matthew Brown and Douglas J. Dempster. "The Scientific Image of Music Theory". In: *Journal of Music Theory* 33.1 (1989), pp. 65–106.

[3]Iannis Xenakis. *Musiques Formelles*. Paris: Editions Richard-Masse, 1963.

[4]Julian Hook. "Uniform Triadic Transformations". In: *Journal of Music Theory* 46.1/2 (2002), pp. 57–126.

where the process of making in both analysis and composition may not be perceived as directly related to the final product anymore, and the effect of changing a specific part of the process is difficult to be anticipated or even perceived in the final product. Finally, many musicians may experience a *sensual distance* due to the abstractness of mathematics. The act of writing formulas is less easily perceived as a physical act than for instance playing the piano and may thus likely be seen as a purely mental activity, detached from the physical world.

The goal of this thesis is to bring mathematical music theory, especially the theory of musical transformations, closer to musicians by diminishing precisely these types of distances. The final product is a musical software that encourages experimentation with mathematical objects and transformations and facilitates understanding by making them available in a more direct and intuitive way to composers, improvisers, and analysts. The software enables its users to observe and interact with their creative processes in a physical way, removed from the abstraction of mathematical formulas.

In this context, it is inevitable to be reminded of a debate that has been going on for almost two decades, based on a few statements by David Lewin that were meant to intuitively explain his theory of musical transformations. Many scholars have expanded on these statements and interpreted them in various ways. Even though the mathematics of transformational theory does not strictly contain what these scholars envisioned, it bears the potential of not only being imagined, but also realized in this way, as this thesis shows.

Lewin saw transformational theory as anti-Cartesian, which means that it has the potential to describe music not from a rational and objective outside perspective, but from the perspective of a musician thinking within the music. Traditional music theory, for Lewin, was concerned with thinking in musical intervals, which consists in

measuring distances between objects that are located in an external space separated from the analyst in a Cartesian dualist way. In contrast, thinking in transformations puts the subject *inside* the music in a anti-Cartesian fashion, as a sort of idealized singer or dancer rather than an traditional analyst or listener observing the music from outside. Lewin illustrated the so-called *transformational attitude* as follows: "if I am *at* s and wish to get to t, what characteristic gesture [...] should I perform in order to arrive there?"[5] and "if I want to change Gestalt 1 into Gestalt 2 (as regards to content, or location, or anything else), what sorts of admissible transformations in my space [...] will do the best job?"

Lewin's formulations in this and related passages are rather vague and have caused a lot of debate. How and why does the transformational attitude oppose Cartesianism? Why can transformations not be observed from outside just as intervals can? Why can intervals not be anti-Cartesian if they are conceived as transpositions and thus specific transformations? What are the gestures that Lewin speaks of and how are they related to transformations? Who is it that performs these gestures? How do these gestures relate to the anti-Cartesianism?

Several scholars have attempted to find answers to these questions. The central difference between Lewin's Cartesian and anti-Cartesian positions is that in the latter someone seems to be actively performing actions instead of passively measuring distances. The subject creates or recreates the music by performing the gestures it takes to transform specific musical entities to other ones.[6] In an earlier article, Lewin said that transformations are names "for 'a way of moving' (from anywhere to somewhere

[5]David Lewin. *Generalized Musical Intervals and Transformations.* New York, NY: Oxford University Press, 1987/2007, p. 159.

[6]For an in-depth explanation and illustration of this, see Satyendra, "An Informal Introduction to Some Formal Concepts from Lewin's Transformational Theory", ch. I. Rings elegantly characterizes this difference: "a transformational analyst is interested in the *how* of a given musical gesture more than the *what* of the resulting pitches and intervals." Steven Rings. "Tonality and Transformation". Ph.D. Thesis. Yale University, 2006, p. 50

else), rather than a relation between fixed points in musical space; it lables [sic] a res fabricans rather than a res extensa."[7] The *fabricans* leads the subject to an entirely different level of reality that builds the basis of the understanding of how the music is or can be fabricated rather than merely witnessed.

Lewin never explicitly specifies who the performer of these actions is. Even though in much of his writing he seems to assume the esthesic position of an analyst or listener who take an active role,[8] he also seems to consider the perspective of a performer, composer, or even the music itself.[9] The movements themselves can then again be taken literally as physical performer gestures,[10] metaphorically as musical gestures present in the music itself,[11] abstract generative gestures performed by a composer, or imaginative gestures a listener performs on Gestalts in the process of understanding certain aspects of the music. Ultimately, for Lewin, the "transformational outlook introduces an attractive kinetic component into theories that suffer from a static character when "dominant" et al. are conceived merely as labels."[12]

[7]David Lewin. "Forte's Interval Vector, My Interval Function, and Regener's Common-Note Function". In: *Journal of Music Theory* 21:2 (1977), pp. 194–237, p. 234.

[8]Such as in David Lewin. "Music Theory, Phenomenology, and Modes of Perception". In: *Music Perception* 3 (1986), pp. 327–92. As Michael Klein points out, GMIT makes few explicit connections between a theory of transformations and the act of composing." Michael Leslie Klein. *Intertextuality in Western art music*. Indiana University Press, 2005, p. 23.

[9]In the introduction to *GMIT* Lewin mentions a composer, besides a singer and a player. Lewin, *Generalized Musical Intervals and Transformations*, p. xxxi.

[10]Steven Rings labels this the *concrete interpretation*. Rings, "Tonality and Transformation", p. 46.

[11]Such as described by Zuckerkandl, who says that musical contexts are kinetic contexts and that hearing music is above all hearing motion. Victor Zuckerkandl. *Sound and Symbol. Music and the External World*. Ed. by Translated by Willard R. Trask. Routledge, 1956.

[12]David Lewin. "A Formal Theory of Generalized Tonal Functions". In: *Journal of Music Theory* 26:1 (1982), pp. 23–60, p. 329. Some scholars believe that Lewin wanted transformational thinking to replace intervallic thinking. For instance Henry Klumpenhouwer. "Essay: In Order to Stay Asleep as Observers: The Nature and Origins of Anti-Cartesianism in Lewin's Generalized Musical Intervals and Transformations". In: *Music Theory Spectrum* 28:2 (2006), pp. 277–89, p. 277-8, and most reviewers he cites. However, Lewin himself does not explicitly express a preference for any of the approaches: "we do not have to choose *either* interval-language or transposition-language; the generalizing power of transformational theory enables us to consider them as two aspects of one phenomenon. Lewin, *Generalized Musical Intervals and Transformations*, p. 160.

What is most anti-Cartesian about this view is that these kinetic actions do not seem to presuppose a rational perspective, but can be understood in a more intuitive embodied fashion, even though they might ultimately be described using mathematical transformations. The subjects seems to do be able to perform gestures without having to think outside of music and without being troubled by non-musical matters. The use of the terms gestures and movements make these actions continuous processes that are comprehensible and observable at every stage through this continuity.

All this seems at first paradoxical in view of the mathematical formalisms that Lewin introduces. Not only do the formalisms themselves appear utterly rational, but the transformations do not seem to account for the continuity Lewin perceives. In other words, the mathematics alone do not justify either Cartesianism or non-Cartesianism. Lewin admits this himself by noting that intervals and transformations can be seen as aspects of the same phenomenon.[13] Specifically, Lewin defines both, intervals and transformations, the same way, as mathematical functions, with the sole difference that transformations do not have to be invertible. Lewin's metaphors are thus not strictly tied to the formalisms of intervals and transformations.[14] On the one hand, we may consider intervals as a special case of transformations, namely transpositions, and are thus able to choose freely between a Cartesian or non-Cartesian perspective. On the other hand, it is perfectly conceivable that an analyst may be observing transformations from an outside perspective, not being actively involved in them.[15]

[13]Lewin, *Generalized Musical Intervals and Transformations*, p. 159/60.

[14]Michael Cherlin. "On Adapting Theoretical Models from the Work of David Lewin". In: *Indiana Theory Review* 14 (1993), pp. 19–43, p. 21.

[15]Julian Hook brings attention to the fact that Descartes considers motion as an object property and would thus look at transformation in the same way. Julian Hook. "David Lewin and the Complexity of the Beautiful". In: *Intégral* 27 (2007), pp. 55–90, p. 173 n. 15. He also emphasizes the importance of objectivity in mathematics and considers Lewin's metaphors unsuitable. ibid., p. 176.

Yet, what is usually not discussed is that it is Lewin's use of graphs and networks in connection with transformations that create the main perceptive difference. While with interval systems we seem obliged to perceive all distances simultaneously, with transformation networks we seem to be able to be more selective and choose to represent the few transformations that appear relevant to us. We have the power to recreate a musical work in one of the many ways in which we may subjectively hear it, or we can bring attention to any of the aspects that seem most relevant to us. Seen this way, the paths in a transformational network may be seen as phenomenological rather than atemporal objective descriptions.

A second problem is the discrepancy between Lewin's strict formalisms and his vague description of the notion of gesture. Even when speaking metaphorically rather than literally, Lewin seems to identify gestures with transformations. However, the mathematical nature of the transformations he uses are in fact plain functions that map arguments to values but do not exactly specify how the argument reaches the position of the value. For instance, if we perform $F\sharp = I_4(B\flat)$, how do we imagine the $B\flat$ to reach $F\sharp$? Is what we imagine any different from what happens during $T_8(B\flat)$? The functions themselves do not describe any of this. In other words, all states between the argument and the value are not specified. Thus, if an arrow in a transformation network stands for a function, its (imaginary) intermediate states are not specified. The continuous appearance of an arrow seems deceiving in relationship to what it does.[16] Of course, it is acceptable to keep the concept of gesture entirely in the mind of the analyst or performer, and to abstractly represent the gesture with a transformation, but this does not seem to be what Lewin envisioned. Further-

[16]Mazzola discusses this in Guerino Mazzola. *La vérité du beau dans la musique.* Paris: Delatour/IRCAM, 2007, p. 155-6 and refers to Gilles Châtelet, who brings attention to the illusory nature of functional arrows. Gilles Châtelet. *Figuring space: philosophy, mathematics, and physics.* Kluwer, 2000. See also Hook, "David Lewin and the Complexity of the Beautiful", p. 175-6.

more, many of Lewin's transformations cannot easily be imagined as gestures. Some transformations, however, can more easily be imagined gesturally than formalized mathematically.[17]

One of the scholars that bring attention to this problem is John Roeder, who suggests the use of animation in order to visualize the processes represented by a transformation network. He maps the space of his example networks to spaces such as a pond or a parquet floor, where lily pads and floor tiles directly correspond to the nodes of the network. Additionally, in order to represent Lewin's "insider", he first introduces an agent such as a fish or a mouse with which the viewers may identify in order to gain the inside perspective themselves. After a few experiments, he concludes that unless "we empathize sufficiently with the agent, we may feel like uninvolved, passive observers."[18] Roeder finally copes with this problem by providing an interactive system, Animation 13, by means of which users can freely travel the edges of a network in first-person perspective and witness the musical outcome, which Roeder considers "perhaps the closest possible realization of Lewin's vision of being "inside" the music".[19]

Nevertheless, there are several limitations to Roeder's approach. Even though he significantly clarifies what Lewin may have meant for an analyst to be inside the music, the solution to the question of gestures seems lacking. He does not sufficiently distinguish between the spaces where the musical objects exist and the space where a transformation network is defined. His animations are mainly concerned with movements along the graph rather than the movements of the musical objects them-

[17]The transformation $MUCH(s)$, for instance, which maps a tone row s into the retrograde inversion the beginning of which overlaps the most possible with the ending of s. Lewin, *Generalized Musical Intervals and Transformations*, p. 8.2.5.

[18]John Roeder. "Constructing Transformational Signification: Gesture and Agency in Bartok's Scherzo, Op. 14, No. 2, measures 1-32". In: *Music Theory Online* 15:1 (2009), p. 11.1.

[19]Ibid., p. 11.3.

selves. The gestures that the agent performs in this space have no relationship to the underlying musical gestures, not even in the metaphorical way Roeder suggests.[20] Furthermore, even though the animations are visually continuous, only the beginning and ending nodes have musical significance. All intermediary places on the arrows have a purely visual existence. In sum, Roeder's systems are mere superficial visualizations of the transformation networks and not of the musical contents described by the networks and their transformations.

Guerino Mazzola also repeatedly addressed the problem of gestures in Lewin[21] and together with Moreno Andreatta provided an extension of his own transformational theoretical approach based on category theory and topos theory.[22] This extension, so-called gesture theory, describes continuous rather than discrete movement by taking the graphs of transformational theory into topological spaces, where the schematic arrows are replaced by continuous curves. The theory is inspired by Mazzola's ontological dimension of embodiment that consists of the three levels of facts, processes and gestures.[23] Lewin's graphs are members of the processual rather than gestural level, for the reasons of discontinuity discussed above. In comparison with Roeder, Mazzola considers gestures to literally take place in the mathematical space where the transformed musical objects are situated.

Despite the problems just described, Lewin's statements and ideas are relevant in terms of the thoughts about the distance between music and mathematics formulated in the beginning of this introduction. The question about anti-Cartesianism is

[20]Roeder thinks that in his example "the performer is not a pianist hitting various keys, but a fish whose gestures metaphorically represent the structure of the pitch changes."Roeder, "Constructing Transformational Signification: Gesture and Agency in Bartok's Scherzo, Op. 14, No. 2, measures 1-32", p. 7.7.

[21]For instance in Guerino Mazzola and Paul Cherlin. *Flow, Gesture and Spaces in Free Jazz. Towards a Theory of Collaboration.* Berlin/Heidelberg: Springer, 2009, p. 67-8.

[22]Guerino Mazzola and Moreno Andreatta. "Formulas, Diagrams, and Gestures in Music". In: *Journal of Mathematics and Music* 1.1 (2007), pp. 21–32.

[23]Mazzola, *La vérité du beau dans la musique*, p. 155-6.

precisely about distance. We may speak of composers, performers, or analysts experiencing a *Cartesian distance* to the music when they do not manage to think inside the music in Lewin's sense. They experience this distance if their concepts, be they mathematical or not, have an existence only outside of the music and fail to appropriately describe what happens in the music. All of the distances introduced above may be seen as specifications or partial distance of such a Cartesian distance. The conceptual distance is minimized when gestures become intuitively understandable in the way Lewin envisioned it, the generative and temporal distances are abolished by the phenomenological perspective, and the sensual distance is no longer perceived if the gestures are seen as embodied, either if they are physical or metaphorical where the subject assumes existence inside the music. As the debate around Lewin's ideas shows, however, it is not mathematics itself that produces these distances, but musical concepts in general.

These ideas and problems gave the initial spark for the work described in this thesis. The outcome, the *BigBang* rubette module for the *Rubato Composer* software, is an exploratory implementation of solutions to these problems. The software's main focus is to apply the principles of transformational theory to composition rather than analysis, poiesis rather than esthesis, and to thereby minimize all types of distances discussed above. It does this by implementing the three levels of Mazzola's ontological dimension of embodiment and allowing users to interact with any of the levels of facts, processes, and gestures. The software also provides functionality that may achieve some of what Lewin had envisioned and may thus be used as an experimental platform for music theorists as well. While Roeder's interactive Animation 13 or the animations created by other scholars presented material based on pre-fabricated networks, this system allows musicians to create transformational networks and the corresponding gestures on the fly, which can be crucial for a deeper understanding of the theory and

supports a more active and free involvement with musical material, perhaps coming as close as possible to Lewinian anti-Cartesianism.

Part I of this thesis introduces the conceptual categories of facts, processes, and gestures, and reassesses recent achievements in music theory, composition, and music informatics in their light. First, any type of transformational theory presupposes specific types of conceptualized musical objects. In Chapter 2, I discuss the types of such musical objects there are by introducing Mazzola's topographic musical ontology. Then, I focus on the dimension of embodiment which explains how such objects may come about or how they may be transformed into each other. Finally, I discuss how the three levels of embodiment are interconnected and how there can be communication between them. In Chapter 3, I briefly outline how three fields or stages of mathematical music theory, set theory, transformational theory, and gestural theory, can be seen as analogous to the three levels of embodiment. In this occasion, I introduce the category-theoretical and gesture-theoretical constructs that this thesis is built upon. Finally, in Chapter 4, I discuss the notions of the processes of composition and improvisation that underlie this thesis and investigate recent developments in computer-assisted music creation, and how they relate to the dimension of embodiment. I will place a special focus on the *Rubato Composer* software for which *BigBang* is designed.

Part II presents the theoretical reasonings behind the software and discusses it from a conceptual point of view. The part is structured according to the three levels of embodiment. Chapter 5 gives a brief introduction to the software. In Chapter 6 I explain the types of facts in terms of musical objects available in *BigBang* and how they can be created, visualized, and sonified. I discuss previous achievements in *BigBang* and comparable predecessors and present many examples of types of musical objects and how they can be represented in *BigBang*. In Chapter 7, I describe how

processes are implemented and visualized, what types of processes there are, and how they can be interacted with in *BigBang*, again with many examples. Chapter 8 introduces the two gestural aspects of *BigBang*: gestural interaction and gesturalization of processes. I connect *BigBang*'s workings to gesture theory and discuss the current ways users can interact with the software in gestural ways.

Finally, Part III is concerned with more technical questions underlying the implementation of *BigBang*, as well as examples of practical uses in composition and improvisation. Chapter 9 discusses relevant aspects of the implementation and outlines the architecture of the software. Even though in the context of this thesis I did not have the opportunity to create elaborate musical works due to time constraints, in Chapter 10 I refer to some of the innumerable smaller examples and sketches I created in order to illustrate some of the various ways of making music with the software. Each of these examples focuses on different musical objects or techniques available in *BigBang*. I briefly explain how they can be created and provide links to SoundCloud and YouTube, where they can be listened to and watched.

Chapter 2

Musical Ontology and Conceptualization

In any musical activity, be it performance, composition, analysis, or listening, we are either confronted with or we are generating concepts that describe a great variety of musical objects. With any form of notation, analytical representation, or compositional constraints, one refers to specific musical entities, often of strikingly distinct nature. Since in this thesis I will be concerned with modeling such objects mathematically and representing them visually and auditorily, as well as with providing ways of interacting with them, it will be helpful to first consider them in a more philosophical way. What are musical works and where do they exist? What kinds of musical objects do musical works really contain and how do these objects get there? What kinds of objects can we hear and what concepts do we create do describe them?

Even though music theorists deal with such questions on a daily basis, they really are subject to musical ontology, the philosophy of musical existence, the goal of which is to identify which musical entities exist, how they can be categorized, and what hierarchies can be established among them. Until recently, musical ontologists have

largely been concerned with musical works and their performances rather than the smaller entities of musical existence that works are made of. Most musical philosophers agree on the fact that musical works are some sort of sound structures, be they abstract or not, eternal or socio-historical. But they rarely attempt to define what these sound structures are made of and are rather concerned with questions of authentic performance, differences between musical works depending on styles, or the Platonist fundamentalist debate, which deals with the question wether or not works exist and whether they are discovered or created by musicians.[1]

Nevertheless, in an obvious sense, many works are composed of several parts, which may in turn be composed of voices, again composed of melodic fragments, and so on. Entities on any such level can be said to have an individual musical existence and are thus worthy of being described and categorized in ontological terms. Even a pitch by itself refers to a greater space of existence that presupposes musical characteristics. Roger Scruton and Stephen Davies are two of the few philosophers who recently addressed the question of the nature of the musical elements that make up musical works.[2] In a nutshell, they take sounds as atomic musical objects and regard them as audible physical objects which only become musical when they relate to other sounds. Scruton calls these musical objects *tones* and claims that for them to exist, it takes relationships such as tonal or rhythmical relationships that are unique to music. He then suggests that in the process of listening we attach metaphors to the movement perceived in the tones themselves and their relationships.[3] Scruton sees these movements as happening in an imaginary rather than geometrical space. In fact, for Scruton, musical objects, their space, and their motion are not even analogous to

[1] Andrew Kania. "The Philosophy of Music". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Spring 2014 Edition. 2014.

[2] Lydia Goehr et al. "Philosophy of music". In: *Grove Music Online. Oxford Music Online*. Oxford University Press, 2014.

[3] Roger Scruton. *The Aesthetics of Music*. Oxford: Oxford University Press, 1997, p. 1-96.

the physical space we live in.[4] Davies, on the other hand, discusses music theoretical entities such as pitches, rhythm, harmony, meter, melody, or instrumentation, and discusses the implications these entities have on the identity of the sound structure of a musical work.[5]

What is striking is that both Scruton and Davies strictly adhere to elements of common-practice music theory to describe smaller musical entities, which seems problematic regarding both music before 1800 and since the twentieth century. Furthermore, their systems lack the description of other ways we may hear or conceive music, in trained and non-trained ways, for instance, as described by Mark DeBellis.[6] Nevertheless, if we think of music theory as an extended field including acoustics and psychology of perception, its concepts describe a great deal of musical objects on any hierarchical level. Not all of these objects may be of the same significance to the composer or listener in any moment of conception or perception. Composers typically do not think of notes as discrete objects but rather as parts of a larger melodic/harmonic gestalt. In a similar way, listeners generally perceive groups of notes with certain relationships rather than individual frequencies or time points. This intermediary level, reasonably efficient and informative, is what is usually referred to the level of *basic categories*.[7]

One of the strengths of transformational theory is that it can be applied any type of musical object, on any hierarchical level. A transformational network may describe higher formal principles as shown in Lewin's second book,[8] but also on a micro-level,

[4]Scruton, *The Aesthetics of Music*, p. 56.

[5]Stephen Davies. *Musical works and performances: A philosophical exploration.* Oxford University Press, 2001, p. 45-71.

[6]Mark DeBellis. *Music and Conceptualization.* Cambridge University Press, 1995.

[7]Lawrence M Zbikowski. *Conceptualizing music: Cognitive structure, theory, and analysis.* Oxford University Press, 2002.

[8]David Lewin. *Musical Form and Transformation: Four Analytic Essays.* New Haven: Yale University Press, 1993.

showing relationships between pitches classes, such as in Klumpenhouwer networks.[9] The work in this thesis builds on a way of modeling musical objects from ground up, in a hierarchical way, using mathematical category theory. Nevertheless, it is a crucial requirement for the software developed in the context of this thesis to offer musicians the possibility to intuitively define and manipulate objects on the level of basic perceptual categories, as will be discussed later on.

First, let us turn to a more open yet systematic characterization of the nature of musical entities, based on both, ontology and semiotics. In discussing it, I will repeatedly refer to other philosophers' work.

2.1 Mazzola's Topography of the Ontology of Music

Many of the notions of musical works and thus also smaller musical entities include a characteristic that we have not yet discussed. David Davies, for instance, sees a work mainly as an intentional outcome of actions performed by a composer.[10] Stephen Davies sees it as more or less "thick" instructions for a successful performance.[11] Others think of a work as the collection of all potential performances. These are all reasonable notions that prove viable depending on what type of work or what aspects of a work one is interested in. However, it seems that each of these notions focus on different perspectives of what might constitute a musical communication process. Scholars interested in musical meaning have developed a branch of music theory concerned with semiotics. It may be claimed that the semiotic level is basic to

[9] David Lewin. "Klumpenhouwer networks and some isographies that involve them". In: *Music Theory Spectrum* 12.1 (1990), pp. 83–120.

[10] David Davies. *Art as Performance*. Malden, MA: Blackwell, 2004.

[11] Davies, *Musical works and performances: A philosophical exploration*.

our understanding of musical ontology. This is what Mazzola does in his ontological topography, first introduced in *Geometrie der Töne*.[12]

Mazzola's system is an extension of Paul Valéry's artistic and Jean Molino's musical *communication* scheme,[13] adopted by Jean-Jacques Nattiez,[14] which includes the three stages of poiesis, trace (neutral level), and esthesis, which purposefully separates the esthesic effect from poietic intention. *Poiesis* includes the intended objects of the originators or senders of the musical message, which include for instance composers, performers, or analysts describing their way of hearing a piece. The *neutral level* includes the work itself in a form such as a musical score, a recording, or informal performance instructions. *Esthesis* is the activity of the receiver, who can be an audience, a theorist, an analytical computer tool, and so on. Interestingly, the three notions of musical works discussed in the previous paragraph each correspond with one of these stages, respectively. The first focuses on the poietic process of the composer, the second on the score as a set of instructions, and the third as all potential performances, which are again potential poietic achievements, depending on the level of freedom a score allows for. It becomes clear from this, that a single work can go through several iterations of this communication process. For instance, composers are typically already involved in a listening or analytical process when examining their own work during their poietic process and the poietic process thus again consists of an encapsulated poiesis/neutral level/esthesis, the neutral level figuring in intermediary versions or sketches of the composition.

Mazzola extended this scheme by first adding two additional dimensions, each of them perpendicular to the dimension of communication, as shown in Figure 2.1.

[12]Guerino Mazzola. *Geometrie der Töne: Elemente der Mathematischen Musiktheorie*. Basel: Birkhäuser, 1990, pp. 1-13.

[13]Jean Molino. "Musical Fact and the Semiology of Music". In: *Music Analysis* 9.2 (1990), pp. 105–56, p.106.

[14]Jean-Jacques Nattiez. *Musicologie générale et sémiologue*. Paris: Christian Bourgois, 1987.

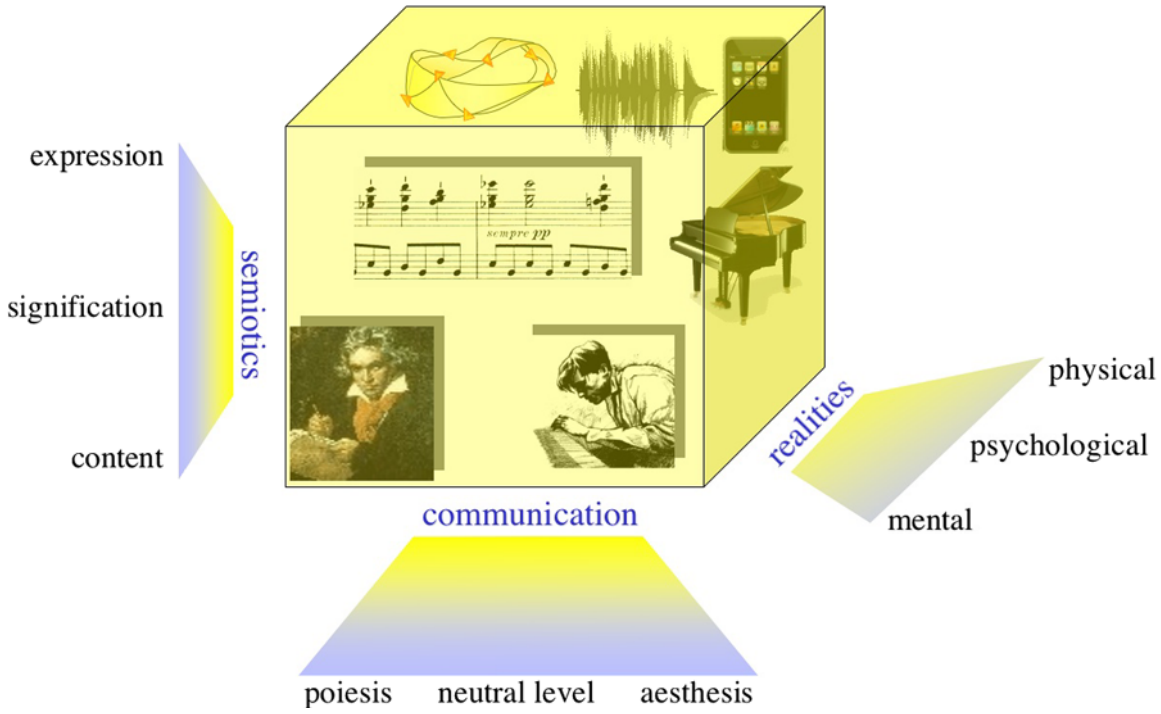


Figure 2.1: Mazzola's three-dimensional topographic ontology.

The *semiotic* dimension includes the three Hjelmslevian stages signifier, signification, and signified. While Nattiez had uniquely focused on the semiotic aspects of the neutral level,[15] Mazzola's model extends the discussion to semiotic existence in both the poietic and esthesic levels. A score or a recorded performance, both elements of the neutral level, are thus not the only imaginable signifiers. Both the composer and the listener may create imaginary signifiers that do not figure in the score, attaching to it their own signified, and may in turn communicate this to another audience.

The dimension of *realities* adds a distinction between physical, mental, and psychological existence. Whereas it is debatable where musical works themselves exist and whether they are physical entities or not, their ultimate purpose is always to decide upon and bring into existence physical musical entities. Mazzola's physical reality level hosts any musical instance that is audible, tangible, or visible, such as

[15]Nattiez, *Musicologie générale et sémiologue*.

scores, instruments, speakers, or performances. The mental reality accounts for musical concepts such as music theoretical concepts or perceptive concepts, as described in the last section. Finally, the psychological level contains any concepts relating to musical expressivity, emotions, and so on.

The iterative recursive nature that I described above for the communicative dimension applies to every dimension and Mazzola sometimes refers to this as the *Babushka principle*. In the semiotic dimension, for instance, we may identify the representation of even a single musical sound as an iterative semiotic procedure. A tone with a frequency of about 440 Hz is what we call A3, which can be represented as a note head with two ledger lines below the treble clef staff, which we can type as `{a3}` in the programming language LilyPond, and so on.

In sum, any musical object, on which ever hierarchical level it may be, has an existence in some, many, or all the nodes of intersection between levels of the three dimensions. A score, for instance, is primarily an element of the physical, neutral, and signifier intersection, whereas the tritone as *diabolicus in musica* primarily exists on the neutral, signified, and mental or psychological intersection, within a certain historical context of meaning.

2.2 The Dimension of Embodiment

In the previous sections, I have repeatedly used the word process, conceiving the work as an evolving entity. Thereby I have faced a major problem with ontology, which most often attempts to capture rigid entities and freeze them in time. However, musical works are arguably much more dynamic entities that may evolve in several stages of reworking, with changing performance practice or with shifting analytical

understanding, or they may even be conceived as open works.[16] Poiesis itself is a process that may take many years, and yet the work exists on the neutral level at every stage along the process and might even be performed.[17] The same is true for esthesis, which in a similar way, cannot be seen as an atomic activity. With each hearing, the process of listening can be different, the listeners may focus on different musical entities, may choose a different way of hearing, or may be influenced by what they heard in earlier hearings or learned by analyzing the score.[18] In a similar way, signification can be seen as a process. Many music theorists argue for a more dynamic understanding of musical works by seeing musical entities as contextual rather than Platonically fixed, emerging from a specific socio-historical context, and evolving with its transfer through history and cultures.[19]

A second point of concern is that processes do not really describe what they claim to describe, as they are still abstracted and discrete in nature, just as are the facts. Musicians often use more continuous terms when they describe actions, as already discussed above, regardless of whether they are describing physical, mental, or psychological matters. The term *gesture* appears in all these contexts. Theodor W. Adorno, for instance, speaks of the score as the "seismographic curves, which the body has left to the music in its gestural vibrations,"[20] in a quite literal sense, envisioning composing as a gestural process, both physically and metaphorically. Score notation is then, for Adorno, an artificial way of conservation that negates gestures and kills music.[21] Renate Wieland, a student of his, sees gestures as having evolved

[16]Pierre Boulez. "Sonate, que me veux-tu?" In: *Perspectives of New Music* (1963), pp. 32–44.

[17]As seen above, Scruton or David Davies see works of music as intended objects resulting from a series of human actions. Scruton, *The Aesthetics of Music*, p. 107. Davies, *Art as Performance*

[18]Already Molino speaks of poiesis and esthesis as processes. Molino, "Musical Fact and the Semiology of Music", p. 105-6.

[19]Jerrold Levinson. *Music in the Moment*. Cornell University Press, 1997.

[20]Theodor W. Adorno. *Zu einer Theorie der musikalischen Reproduktion*. Frankfurt am Main: Suhrkamp, 2001, p. 247.

[21]Ibid., p. 235/44.

from a specific meaningful action context and being abstractions thereof, they still exist in a gestural coordinate space.[22] Robert Hatten, who emphasizes the importance of developing a formal theory of musical gestures as a branch of music theory, defines gesture as a "communicative (whether intended or not), expressive, energetic shaping through time (including characteristic features of musicality such as beat, rhythm, timing of exchanges, contour, intensity), regardless of medium (channel) or sensory-motor source (intermodal or cross-modal)."[23] These scholars define gestures independently of a semiotic context and of the process of signification, in contrary to the popular notion of gestures as a specific carriers of meaning. This does not mean that they cannot carry meaning but they are often abstracted from it.

In a more general sense, gesture is increasingly regarded as not only supporting our thinking processes, but literally as a means to learn to think. Susan Goldin-Meadow, for instance, sees gestures as crucial in a child's process of learning to think mathematically: "Advances in mathematical reasoning are very likely to come first in gesture – and they do. [...] Do new ideas always come first in gesture, regardless of domain?"[24] Neuroscientists Gentilucci and Corballis suggest that language and gesture evolved as a joint communication system, which can still be seen in the accompanying gestures speakers make.[25] This coincides with ideas in recent French philosophy such as Maurice Merleau-Ponty's, who repeatedly refers to words as gestures.[26] Mazzola finds in this and the writings of Gilles Deleuze and Charles Alunni, among others, the notion of gestures as *presemiotic* rather than semiotic entities.[27] In sum, as in many

[22]Cited and explained in Guerino Mazzola. *Musical Performance: A Comprehensive Approach: Theory, Analytical Tools, and Case Studies*. Berlin Heidelberg: Springer, 2011, p. 121f.

[23]Robert Hatten. *Interpreting Musical Gestures, Topics, and Tropes*. Indiana University Press, 2004.

[24]Susan Goldin-Meadows. *Hearing Gesture: How Our Hands Help Us Think*. Harvard University Press, 2003.

[25]Maurizio Gentilucci and Michael C Corballis. "From manual gesture to speech: A gradual transition". In: *Neuroscience & Biobehavioral Reviews* 30.7 (2006), pp. 949–60.

[26]Maurice Merleau-Ponty. *Phénoménologie de la perception*. Gallimard, 1945, p. 211f.

[27]Mazzola and Cherlin, *Flow, Gesture and Spaces in Free Jazz. Towards a Theory of Collaboration*,

other fields, in music cognition scholars have increasingly let go of Cartesian dualism and considered musical experience as an embodied process in which the listener directly engages with the music in a way directly related to physical activity.[28]

Inspired by the French school, Mazzola extended his three-dimensional ontological topography by adding a fourth dimension after realizing that his earlier model only included facts, in other words everything that is the case in a Wittgensteinian sense.[29] The process of creating or understanding something or the process of evolution could not be situated within the previous three-dimensional ontological system, such as for instance improvised music or gradual understanding in a phenomenological sense. Mazzola decided to add the so-called *dimension of embodiment*, which again includes three levels: facts, processes, and gestures, as shown in Figure 2.2.

The level of *facts* contains everything that is the result of a process, be it final or intermediary.[30] In composition, for instance, it includes any record of a work or a performance, either published in the form of a score or a recording or existing as an intermediary sketch or version. Evidently, not all results of processes or partial processes survive in material form, especially if they are intermediary stages of thought processes. Other examples of facts are any stage of an analytical process of a work or stages of psychological reaction in the listening process.

On the level of *processes* we find descriptions of how specific facts are created and explanations of how we can get from one fact to another. These exist in the form of operations or actions manipulating musical material, possibly to be broken down to

p. 74.

[28]See for instance Marc Leman. *Embodied Music Cognition and Mediation Technology.* Cambridge: MIT Press, 2007, p. 3f.

[29]Mazzola, *La vérité du beau dans la musique*, pp. 154-6. Mazzola and Cherlin, *Flow, Gesture and Spaces in Free Jazz. Towards a Theory of Collaboration*, p. 32.

[30]Mazzola usually speaks of facts being the final product of a process. However, with any partial process being again a process, the outcome of any partial process is again a fact. Thus, at any intermediary state of a process, we must find a fact.

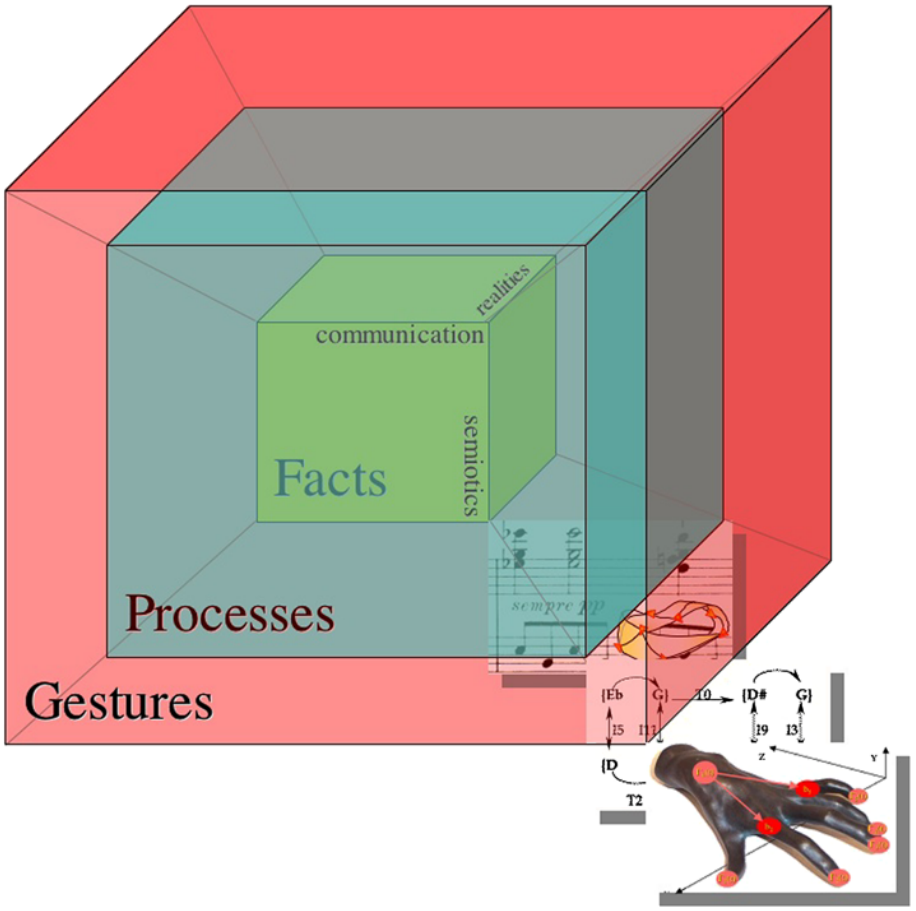


Figure 2.2: Mazzola's extension of the ontology cube to a hypercube by introducing the dimension of embodiment.

any level of detail, until atomic operations are reached. These actions are typically called transformations, as it is common in process philosophy.[31] Processes also exist on a more abstract level as diagrams that may be applied to a variety of facts. For instance, if a musical theory gives us a recipe on how to extract certain analytical facts from a musical work, it is typically successfully applicable to an entire family of works in order to be viable. On the other hand, a compositional theory describing a process may lead to a great variety of pieces when applied to various musical material.

Finally, the level of *gestures* accounts for what processes fail to describe. Gestures

[31] Johanna Seibt. "Process Philosophy". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Fall 2013. 2013.

are continuous entities that have a temporal existence. However, they are not bound to the physical level. Gestures may exist as mental representations, psychological signifieds, and so on,[32] and they are independent from semiotics, as seen above. While processes are abstract descriptions of an operation or action, both the realization and the comprehension of a specific process can typically happen in various ways, depending on the means and capacities of the individual.[33] For instance, the result of an action may simply be a musical transposition, but the ways in which this can be achieved, vary greatly.

In sum, the dimension of embodiment represents three increasing degrees of embodiment, from facts to gestures. While processes still relate to specific gestures by being disembodied representations of them, facts are simply the results and we need a certain amount of analytical capabilities to reconstruct the originating gestures that made the facts. This relates strongly to the discussion in the introduction of this thesis. Both recent philosophy and psychology have come to the conclusion that much of our understanding of the world comes from an anti-Cartesian understanding of ourselves as part of the world, rather than beings reasoning about it from outside, or even ones based on given metaphysical constants. The way we make and understand music is to a great extent determined by an understanding of temporal activities, such as the physical gestures of a performer or the mental gestures of a composer, which we may perceive in any existing form of music. This does of course not mean that everything in music is gestural. For instance, abrupt changes or ironic juxta-

[32]Mazzola associates gestures with the activity of *making*. This of course also includes the comprehension and reenactment of how something was made. Similarly, recent neurological research has show that when someone is observing the activity of someone else, their so-called mirror neurons act in precisely the same way as the neurons of the person who is performing the activity.

[33]This compares to what Scruton says about actions and events. The former contains an intentionality, whereas the latter is a means to achieve it. Actions can be performed using different events, *but* the same event can also be used to perform different actions. Scruton, *The Aesthetics of Music*, p. 107. Gestures, for Scruton and many other musical scholars, are purely metaphorical movement.

positions may precisely play with the absence of gesture and continuity. However, based on recent findings in the psychology of perception,[34] we claim that the way even such juxtapositions are understood, is by establishing a relationship between the discrete juxtaposed parts, and performing an imaginary comparative gesture in a mental space.

2.3 Communication Between the Levels of Embodiment

In early descriptions of the new ontological dimension, Mazzola characterized the procedures necessary for the communication or translation between these levels in the direction of increasing abstraction. In order to become processes, gestures need to be disembodied and schematized, and in turn, processes are evaluated and facts dissected from them.[35]

In later publications inspired by the work for this thesis,[36] we described this communication in a more detailed way and introduced the three procedures formalizing, factualizing, and gesturalizing, which also describe communication in the other direction, from processes to gestures (see Figure 2.3):

- *Formalizing* consists in distilling the essence of a gesture and representing it in an abstract, schematized way. For instance, a series of performer gestures may

[34]Roger N Shepard and Lynn A Cooper. "Mental Images and Their Transformations". In: *MIT Press*. 1986.

[35]Mazzola and Cherlin, *Flow, Gesture and Spaces in Free Jazz. Towards a Theory of Collaboration*, p. 33. These translations already appear in Gilles Châtelet's writings: "A diagram can immobilize a gesture, put it to rest long before it hides itself within a sign, and this is why the contemporary geometers or cosmologists love diagrams and their power of preemptive evocation". Châtelet, *Figuring space: philosophy, mathematics, and physics*, p. 9-10.

[36]Florian Thalmann and Guerino Mazzola. "Poietical Music Scores: Facts, Processes, and Gestures". In: *Proceedings of the Second International Symposium on Music and Sonic Art*. Baden-Baden: MuSA, 2011.

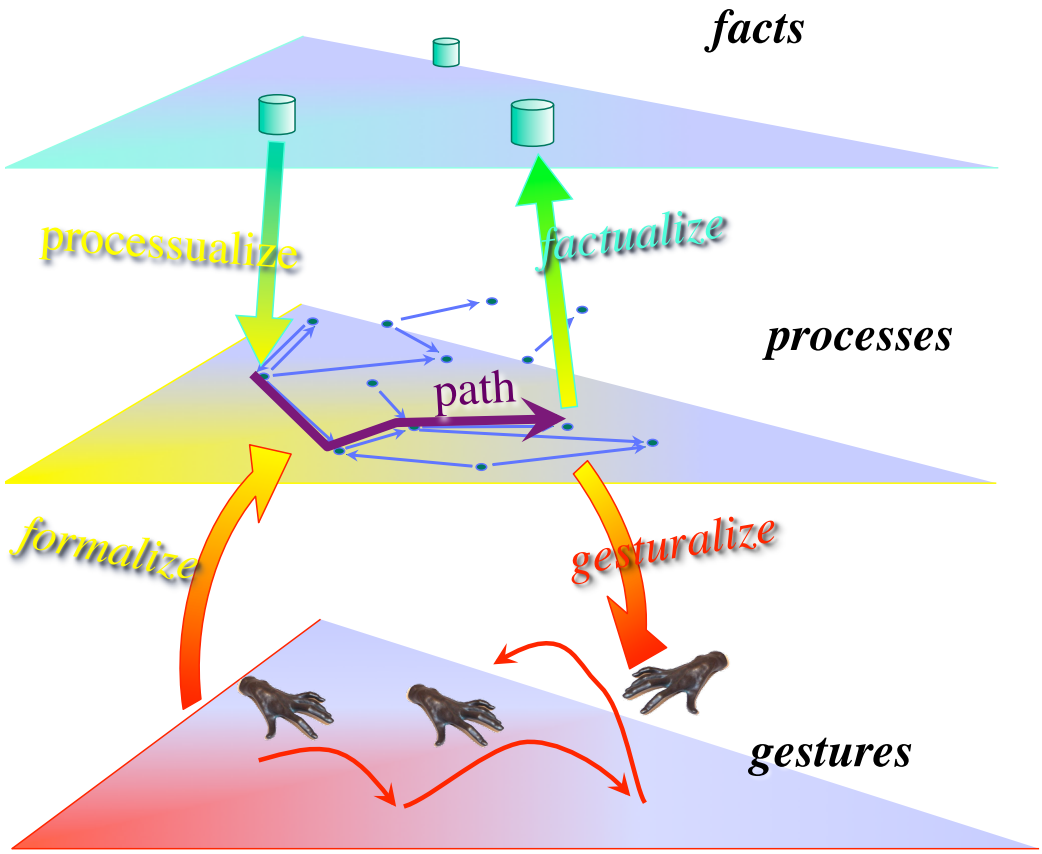


Figure 2.3: The three levels of embodiment and the arrows symbolizing communication between them.

be represented as a diagram of gestures in the manner of a flow chart. Or, we may attempt to find a mathematical function for a transformation imagined in a gestural way.

- *Factualizing* is the procedure of obtaining facts from processes, analogous to determining the resulting dish from a cooking recipe. This is for instance done by composers who complete another composer's work starting from sketches and processual drawings. On the other hand, the application of a standardized analytical procedure to a piece also leads to factual results and can thus be seen as an instance of factualizing.

- *Gesturalizing* recreates gestures from processes by defining and interpolating subprocesses. Any reenactment or understanding of an abstracted process presupposes this procedure. For example, it is a substantial part of the listening process. When we listen to a recording we may variously reconstruct performer gestures, if we are familiar with the instrument, conductor gestures, in case of larger groups, composer gestures, if we are familiar with the composition process, or psychological gestures evoked by the ambiance or extra-musical information we hold about the event.

Missing in this description is a fourth procedure, doubtlessly the most difficult and ambiguous of all:

- *Processualizing* consists in deriving processes from facts, which may be done in innumerable ways and which lies at the center of music analysis. It does not merely include the reconstruction of the compositional process based on the final material existence of a work and possible intermediary documents or sketches, but might for instance equally consist in the construction of a speculative or personal hearing process, or the derivation of procedurally related analytical levels.

In this thesis, I propose solutions for the implementation of the first three procedures which are rarely discussed in computational music theory. Numerous recent activities have focused on the implementation of analytical software components, including ones for *Rubato Composer*.[37]

[37]Ruhan Alpaydin and Guerino Mazzola. "A Harmonic Analysis Network". In: (forthcoming).

Chapter 3

The Paradigm Shift towards Gestures in Music Theory and Analysis

Music theory, especially mathematical music theory, provides a great variety of concepts that can be seen as corresponding to the three ontological levels of facts, processes, and gestures discussed in the previous chapter. To a certain degree, we can even identify a gradual transition from the former to the latter, most probably inspired by developments in mathematics, philosophy, psychology, computer science, and other fields. Many fields have seen such a transition that could be characterized as a paradigm shift, sometimes called the topological turn, even if often the concepts corresponding to the three levels coexist and are jointly used in theoretical treatises and analytical research.

The focus in this chapter is to find and describe some of these concepts used in music theory, including the formalisms underlying the work of this thesis, and to discuss their relationship to each other and to the levels of facts, processes, and

gestures.

3.1 Facts and Set Theory

Mathematical set theory, group theory, and combinatorics were the inspiration for a theoretical toolbox now accepted as a standard for the analysis of atonal and sometimes tonal music. Musical set theory was initiated by Howard Hanson and Allen Forte, in reaction to Milton Babbitt's mathematical twelve-tone theory.[1] The application of musical set theory consists in identifying sets of musical objects, relating these sets to each other, and categorizing them into set classes based on intervallic content. The musical objects set theory deals with can vary greatly, they can be onsets, durations, beat-classes, timbres,[2] but most often they consist in pitches or pitch-classes. Musical sets are almost always unordered sets written in curly brackets, e.g. $\{3, 4, 7\}$ for the set including the pitch-classes corresponding to $D\sharp, E$ and G.[3] Sometimes, however, when the order in which the elements appear is considered important, such as in twelve-tone theory, one deals with ordered sets, or sequences, often written as e.g. $\langle 3, 4, 7\rangle$.

From an ontological perspective, the elements of sets in musical set theory are nothing beyond what we considered to be musical facts above, on a very low level of

[1] Howard Hanson. *Harmonic Materials of Modern Music: Resources of the Tempered Scale*. New York: Appleton-Century-Crofts, 1960; Allen Forte. *The Structure of Atonal Music*. New Haven and London: Yale University Press, 1973. There were historical precedents such as Wolfgang Graeser, who described contrapuntal form as "a set of sets of sets of notes." Wolfgang Graeser. "Bachs Kunst der Fuge". In: *Bach-Jahrbuch* (1924), p1ff. Many others have expanded on set theory, such as Robert Morris who generalized it and applied it to composition, or Lewin and Mazzola who both generalized it (see below). Robert D Morris. *Composition with pitch-classes: a theory of compositional design*. New Haven: Yale University Press, 1987.

[2] Gary Wittlich. "Sets and Ordering Procedures in Twentieth-Century Music". In: *Aspects of Twentieth-Century Music*. Ed. by Gary Wittlich. Englewood Cliffs, New Jersey: Prentice-Hall, 1975.

[3] Pitch-classes are commonly denoted by numbers from 0 to 11, where 0 is often equated with C or a pitch center.

musical hierarchy. They are characteristics of musical objects existing in the same musical space, most commonly pitch-class space (mathematically $\mathbb{Z}_{12}$) or chromatic pitch space (mathematically $\mathbb{Z}$), just as the objects they correspond to exist in the same space, for instance a score.

Set theory also establishes relationships between these ordered sets of objects in order to determine some of the structural relationships that may have informed the compositional process, or structural relationships that can be heard. The two most common relationships established between musical sets, most often constituted of pitch-classes, are *transposition* $T_n(x) = x + n$, where n refers to the amount of semitones by which T transposes, and *inversion* $I_n(x) = 2n - x$, which inverts around n. For example, $T_4(\{3, 4, 7\}) = \{7, 8, 11\}$. All sets related by transposition or inversion belong to the same set class and share the same intervallic content. Other relationships often found in set theory include the set-theoretical *complement* and the *multiplication*, where elements of two sets are multiplied pairwise and united. Another important relation on sets is the *Z-relation* that relates two sets with the same interval content but are not members of the same set class. For ordered sets, various *permutations*, including *retrograde*, are often used.

Many of these relationships can be seen as instances of processes between facts of the same type, especially since they often consist of transformational relationships. Transposition, inversion, and retrograde, for instance, are defined as functions that transform one set into another, and multiplication transforms two sets into a third one. However, not all scholars agree on seeing them as transformations. As we have seen in the introduction of this thesis, Lewin considers the first part of his book, which is a generalization of set theory, as mere measuring intervals between musical entities, rather than transforming them. What may have caused this understanding is that relations between sets are rarely considered directional. A set is rarely thought

of being causally transformed into another one, but rather existing at the same time as the sets it relates to. From this perspective, what theorists do when they use set theory is show what relevant relationships can be found between facts.

3.1.1 Generalized Set Theory

Lewin expanded on set theory by introducing *generalized interval systems (GIS)*, a special case of which are regular set theory statements. A GIS consists of a set[4] S of musical objects, which Lewin calls space, a group[5] $IVLS$ of intervals or relationships, and a function int, which assigns to a pair $< s, t > \in S$ a member of $IVLS$, thus $int(s,t) = i$ with $i \in IVLS$. Traditional set theory is embedded in this construct if we choose $S = \{0, 1, \ldots, 11\}$, $IVLS = \{T_n, I_n\}$, which forms a group. The only formal difference is that technically, the function int is not defined for pairs of subsets of S, but for pairs of its elements, which is a significant restriction. In the first part of *GMIT*, Lewin introduced and illustrated the use of many other types of objects, many of which had already been used by set theorists before, such as time points, diatonic pitches, duration-classes, and so on. However, he also frequently combined them into products, which led to types of musical objects new to mathematical music theory, such as time spans, Klangs ($pitch \times quality$), which initiated the field of neo-Riemannian theory, or even Schenker objects ($pitch \times degree \times level$).

Along with these definitions, Lewin also introduced generalizations of some other constructs often used in set theory. For instance, the interval vector, which expresses

[4] Lewin calls sets families.

[5] A *group* is a set G along with a binary composition operation $* : G \times G \to G$ where:

- $(g * h) * k = g * (h * k)$ for $g, h, k \in G$ (associativity)
- there is an $e \in G$ for which $e * g = g * e = g$ for all $g \in G$ (neutral element)
- there is an $h \in G$ with $g * h = e$ for every $g \in G$ (inverse)

A group is abelian, or commutative, if $m * n = n * m$ for all $m, n \in G$.

a set's or set class' interval content, was replaced by the function $IFUNC(X, Y)(i)$, which determines the occurrences of a certain interval i between two sets (or Lewin's spaces) X, Y. For $X = Y$ we obtain the $i - th$ element of an interval vector, for $i = 1, \ldots, 6$. Another example is the embedding number $EMB(X, Y)$ that counts the number of occurrences of elements of the set class of X in set Y.[6]

3.2 Processes and Transformational Theory

As seen in the introduction, the second part of Lewin's *GMIT* introduces constructs of a significantly distinct nature. We will first briefly review them, before describing the related formalisms that this thesis is concerned with.

3.2.1 Transformation Graphs and Networks

Simply put, a Lewinian *transformation network* is a GIS enhanced by the visual elements of a directed graph. It again consists in a set S of musical objects along with the a set of transformations SGP, this time forming a semigroup,[7] which means that the transformations do not need to have inverses, as opposed to $IVLS$. The visual elements of the graph are represented by $NODE$ and $ARROW$, two sets. Finally, there are two functions $CONTENTS$ and $TRANSIT$ assigning the nodes of the graph to elements of S and the arrows to elements of SGP, thus $CONTENTS(n) = s$ with $n \in NODES, s \in S$ and $TRANSIT(a) = t$ with $a \in ARROW, t \in SGP$. Lewin also defines *transformation graphs*, which are networks without musical objects or without any contents in their nodes, i.e. without S and $CONTENT$. All transformations in SGP are functions $f : S \to S$ that musical objects in S onto other ones. For

[6] Lewin, *Generalized Musical Intervals and Transformations*, ch. 5-6.

[7] A *semigroup* is similar to a group (see note 5), but it only needs to fulfill associativity and needs to include neither a neutral element nor inverses. Thus all groups are also semigroups, but not vice versa.

bijective transformations, which means that all objects in S can be reached by the transformation and none are reached through different elements of S, Lewin uses the term *operation*.[8]

Transformational theory, even if its systems are defined in a way similar to GISes, has an entirely different goal according to Lewin. The directed graphs that are central to the analytical method introduce an unmistakable causality. The facts that transformations do not have to be bijective, that they may form a semigroup, and not a simply transitive group as it is the case for GISes, add to this causality or directionality. Of course, the causal relationships do not necessarily have to be thought to exist in the music itself. They can stand for a way of hearing the analyzed piece, a hypothetical description of how it may have been composed, or an analytical process of the transformational theorists themselves. Nevertheless, with the added causality, we have now no other choice than speaking of processes, wherever these may take place. In sum, transformational networks are processes composed of atomic transformations that illustrate how a selection of musical objects can be transformed into each other. They are highly selective forms of visualization that do not have to show everything that is happening in the music. Transformational graphs, on the other hand, are abstracted processes that exist independently of specific objects, but that can be concretized by inserting a specific object in the source node of the graph.

3.2.2 Transformations in the Category of Modules

A slightly different version of transformational theory was developed by Mazzola contemporaneously to Lewin. Instead of using sets, groups, and graphs as such, it is based on higher level mathematical theories, category theory and topos theory. Category theory generalizes the way we handle many different mathematical constructs

[8] GISes use only operations.

by providing a way to abstractly express the concepts used by all of them: objects and mappings, which are called morphisms.[9] Many subfields of mathematics can be expressed in terms of categories, such as set theory, group theory, graph theory, metric spaces, or formal logic. For instance, in the category of sets the objects are sets and the morphisms are functions, just as they are used in GISes, and in the category of groups, the objects are groups and the morphisms are group homomorphisms.

Most of the early uses of Mazzola's theory are based on the category of modules[10] **Mod**, which are generalizations of vector spaces. Early examples were simply based on n-dimensional free modules and their direct sums. If a one-dimensional module over the integers modulo 12 $\mathbb{Z}_{12}$, for instance, is taken to represent pitch-classes, then different morphisms can be used to create different trajectories in pitch space, such as the circle of fifths or the chromatic circle.[11] In a similar way, Mazzola produces a classification of all possible chords in $\mathbb{Z}_{12}$ by calculating the isomorphy classes based on repeated multiplication on a monoid based on a single morphism, e.g. the major

[9] More precisely, a *category* $\mathbf{C}$ is a collection of so-called morphisms $f, g, h, \ldots$ along with a partial composition $\circ$, which lets us combine some f, g into a new morphism $f \circ g$ of $\mathbf{C}$. An object, or identity, in $\mathbf{C}$ is a morphism e for which $e \circ f = f$ and $g \circ e = g$. The following has to apply:

- if $f \circ g$ and $g \circ h$ are both defined, $(f \circ g) \circ h$ is defined.
- if either $(f \circ g) \circ h$ or $f \circ (g \circ h)$ is defined, both are defined and denoted $f \circ g \circ h$.
- a morphism f has two identities. e_L and e_R such that $e_L \circ f = f$ and $f \circ e_R = f$.

[10] A left *R-module* is a triple $(R, M, \cdot : R \times M \to M)$, where $(R, +_R, *_R)$ is a ring (see below) of *scalars*, and $(M, +)$ is an abelian group of *vectors*. $\cdot$ is called the scalar multiplication. The following properties apply:

- $1_R \cdot m = m$ for all $m \in M$
- $(r +_R s) \cdot m = r \cdot m + s \cdot m$,
  $r \cdot (m + n) = r \cdot m + r \cdot n$,
  $r \cdot (s \cdot m) = (r *_R s) \cdot m$, for all $r, s \in R$ and $m, n \in M$.

A *ring* is a triple $(R, +, *)$ where $(R, +)$ is an abelian group (see note 5) and $(R, *)$ is a commutative monoid, and for all $r, s, t \in R$, $x * (y + z) = x * y + x * z$, and $(x + y) * z = x * z + y * z$ (distributivity).

A *monoid* is a semigroup with a neutral element, or a group without inverse.

[11] Guerino Mazzola. *Gruppen und Kategorien in der Musik: Entwurf einer mathematischen Musiktheorie*. Berlin: Heldermann Verlag, 1985, p. 6.

triad is produced using $T^7.3(x), x \in \mathbb{Z}_1 2$, which represents a multiplication by 3 followed by a transposition (addition) by 7. So for $x = 0$ we get $T^7.3(0) = 7$ and $T^7.3(7) = 4$, then $T^7.3(4) = 7$, etc.[12] In another example, a direct sum module $\mathbb{Z}^3 \oplus \mathbb{Q}$ over rational numbers can express a space the vectors of which are notes in Euler pitch space $\mathbb{Z}^3$ and onset space $\mathbb{Q}$.[13] Or the vectors of a four-dimensional module over the real numbers $\mathbb{R}$ can represent notes with onset, duration, loudness, and duration. On this module we can then, for instance, define a set of morphisms that allow us to perform geometrical transformations.

What distinguishes this theory from Lewin's is that the musical objects themselves exist in a space structured independently from the morphisms defined on the space. The objects can be added to each other to form new objects and they can be scaled by being multiplied by scalars. Lewin's objects, in turn, the elements S, have no relationship to each other except the produced by the semigroup of transformations SGP. S is thus not really a space as Lewin calls it. Secondly, in the category of modules **Mod** we have the possibility to define a great variety of morphisms that surpass the capabilities of the functions in SGP.[14]

3.2.3 Denotators and Forms

Starting in 1993, Mazzola extended his theory by introducing a further way of creating musical objects. With the goal of allowing for data structures common in logic and

[12] Mazzola, *Gruppen und Kategorien in der Musik: Entwurf einer mathematischen Musiktheorie*, p. 30/2.

[13] Mazzola, *Geometrie der Töne: Elemente der Mathematischen Musiktheorie*, p. 80.

[14] More detailed analyses of how Lewin's constructs differ from category theory can be found in John Rahn. "Cool tools: Polysemic and non-commutative Nets, subchain decompositions and cross-projecting pre-orders, object-graphs, chain-hom-sets and chain-label-hom-sets, forgetful functors, free categories of a Net, and ghosts". In: *Journal of Mathematics and Music* 1.1 (2007), pp. 7–22, p. 12, and Guerino Mazzola and Moreno Andreatta. "From a Categorical Point of View: K-nets as Limit Denotators". In: *Perspectives of New Music* 44.2 (2006).

information theory, he introduced the concepts of *forms* and *denotators*.[15] The former generalize the spaces available in **Mod** by allowing logical combinations of various spaces using standard types corresponding to logical *AND*, *OR*, and collections. These spaces can be built from low-level basic spaces, called **Simple** forms, which can hold any space in **Mod**. For instance, we can define a **Simple** form *EulerPitch* and equip it with the Euler pitch space defined above as follows:[16]

$$EulerPitch : .\mathbf{Simple}(\mathbb{Z}^3)$$

From there, we can combine forms using the two form types **Limit** and **Colimit**, the former of which lets us built product spaces (logical *AND*) and the latter co-product spaces (logical *OR*). For instance, if we wish to build a note that includes an Euler pitch, we may define:

$$EulerNote : .\mathbf{Limit}(Onset, EulerPitch, Loudness, Duration),$$
$$Onset : .\mathbf{Simple}(\mathbb{Q}),$$
$$Loudness : .\mathbf{Simple}(\mathbf{CHR}),$$
$$Duration : .\mathbf{Simple}(\mathbb{Q}).$$

We end up with a four-dimensional product space, one of the dimensions of which is again three-dimensional. This means that every note needs to have an onset, an Euler pitch, a loudness, and a duration. Onset and duration are defined as rational

[15] Guerino Mazzola. *The Topos of Music. Geometric Logic of Concept, Theory, and Performance.* Basel: Birkhäuser, 2002, p. 47ff.

[16] The format I use to define forms in this thesis is called *Denotex*. Any form is defined as

$$Name : .Type(Coordinator).$$

Name can be freely chosen, *Type* is either **Simple**, **Limit**, **Colimit**, or **Power**, and the coordinators are other forms or a module in **Mod**, depending on *Type* ibid., p. 1143.

numbers, which is beneficial for representing the Western rhythmical notation based on ratios, and loudness is based on **CHR**, the module of character strings, with which we can express values such as *f, mp*, or *ppp*. If we wanted to allow rests, too, we could define

$$EulerNoteOrRest : .\mathbf{Colimit}(EulerNote, Rest),$$
$$Rest : .\mathbf{Limit}(Onset, Duration),$$

which gives us a coproduct space of *EulerNote* and *Rest* and would let us decide for each musical object in this space wether it is a note or a rest. Finally, the form type **Power** allows us to make sets of objects. If we wish to have several notes and rests, we need to embed the space in a powerset as follows:

$$EulerScore : .\mathbf{Power}(EulerNoteOrRest)$$

So far we have seen how to define spaces for musical objects. Now how do we create the objects themselves? For this, we need to define specific values in each of the form spaces involved. An object in a form space is called *denotator*. Each **Simple** denotator must specify an element in the module associated with it. For instance, in order to create a specific onset, according to our definition above, we need to specify a vector in the module of rational numbers $\mathbb{Q}$. For this, we write:[17]

$$onsetAtBeat2 : @Onset(1/4)$$

[17] Again, the notation here conforms with Denotex (see note 16). A denotator is defined as

$$Name : M@Form(Coordinates),$$

where $Name$ is again an arbitrary name, M is the address (introduced later), $Form$ is a defined form, and $Coordinates$ are other denotators of the necessary coordinator forms.

The denotator we just defined is an onset at a quarter note into the piece. An entire *EulerNote*, a **Limit** denotator, would then have to be defined by specifying all of its coordinators (*Onset*, *EulerPitch*, *Loudness*, and *Duration*). For a **Colimit** we have to define exactly one of its coordinators, so an *EulerNoteOrRest* needs either a *EulerNote* or a *Rest* denotator. Finally, a **Power** denotator can hold an arbitrary number of its coordinators. In sum, here is how we can define a sample *EulerScore*:

$$
\begin{aligned}
&twoNoteScore : @EulerScore(noteOne, shortRest, noteTwo),\\
&noteOne : @EulerNoteOrRest(note1),\\
&note1 : @EulerNote(onset1, pitch1, loudness1, duration1),\\
&onset1 : @Onset(0),\\
&pitch1 : @EulerPitch(1, 0, -1),\\
&loudness1 : @Loudness(sfz),\\
&duration1 : @Duration(1/4),\\
&shortRest : @EulerNoteOrRest(rest1),\\
&rest1 : @Rest(onsetAtBeat2, duration1),\\
&noteTwo : @EulerNoteOrRest(note2),\\
&note2 : @EulerNote(onset3, pitch1, loudness2, duration1),\\
&onset2 : @Onset(1/2),\\
&loudness2 : @Loudness(ppp),\\
&pitch2 : @EulerPitch(-1, 1, 1),\\
&duration2 : @Duration(3/2)
\end{aligned}
$$

Figure 3.1 shows a noted version of this example, where we assume an Euler tuning
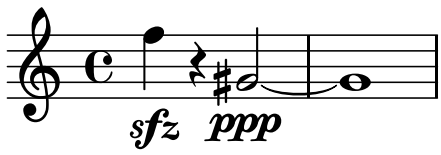


Figure 3.1: The brief *EulerScore* defined above, in staff notation.

space with $a4$ as a reference pitch. $(1, 0, -1) \in \mathbb{Q}^3$ is therefore $f5$ and $(-1, 1, 1)$ $g\sharp 4$. Note that *noteOne* and *shortRest* share the same duration. Also, I reused the denotator named $onsetAtBeat2$ defined above.

These denotators can then be transformed just as we saw it above for module elements, using morphisms. The formalisms of denotators and forms introduce several constructs and possibilities that are not available with plain group or module theory. In contrast with Lewin's transformational theory, forms can describe the logical constructions of coproducts, using **Colimit** as seen in the example, that allow for selection among various options. Furthermore, while above, sets formed the spaces themselves, as in Lewin's S or the sets of vectors in Mazzola's module spaces, **Power** brings sets to the level of musical objects, which means that we can really speak of transforming a set of musical objects as such. In Lewin's theory, we cannot technically define transformations between sets of objects that are all in the same space. Each node of a network can only host a single object in S. The only workaround we have is to define a product space $S \times S \times \ldots S$, a limit in category-theoretical terms, which however only lets us make sets of a certain cardinality.[18]

The category-theoretical approach also allows us to do much more. Even though these possibilities will not be fully taken advantage of in this thesis, I will describe them briefly. For instance, we can define morphisms between different kinds of ob-

[18]These differences are formally discussed in Mazzola and Andreatta, "From a Categorical Point of View: K-nets as Limit Denotators".

jects, each a different member of the category **Mod**. This way, we can relate pitch constellations to rhythms, rhythm to meter, etc, without having to create high-level objects that encapsulate all parameters at once. If we decided to create a form that describes timbre, we could identify relationships between its denotators and denotators of an *EulerScore*, simply by defining an appropriate morphism. In other words, the nodes of a transformational network could contain entirely different musical objects. A step further, we can also introduce *functors*, which are morphisms between categories. If we decided to work with other objects than modules, for instance directed graphs, we could transform the objects in the category of digraphs **D** into objects in the category of modules **Mod**, thus transforming graphs into spaces. Even on a higher level, such functors can be mapped into each other using so-called *natural transformations*.

The system of denotators and forms do implement some functorial language, based on the Yoneda lemma, which replaces objects by arrows, by making them functors. They do in fact, for now, work with the category $\mathbf{Mod}^{@}$ of contravariant functors $Fu : \mathbf{Mod}^{\mathbf{opp}} \to \mathbf{Sets}$. Each denotator in $\mathbf{Mod}^{@}$ has a so-called *address* in another arbitrary module. The coordinate of a **Simple** denotator d is thus an element of $A@Fun(F)$, i.e. a diaffine module homomorphism $f : A \to M$, where F is the form of d and $M = Fun(F)$, a module and the coordinator of F. For instance, we could have defined our denotator *twoNoteScore* to carry an address in $\mathbb{Z}$, which would have enabled us to index its notes and rests and formally create permutations of them.[19] This, however, will not be relevant in the context of this thesis, as we work uniquely with *0-addressed* denotators, the address of which is $\mathbb{Z}^0$, the 0-dimensional module

[19]For this, we would have had to write

$$twoNoteScore : \mathbb{Z}@EulerScore(noteOne, shortRest, noteTwo)$$

etc. See note 17 above. In Denotex, we can drop M for 0-addressed denotators, which is why we wrote $twoNoteScore : @EulerScore(noteOne, shortRest, noteTwo)$ above.

over the real numbers $\mathbb{Z}$, which are analogous to members of **Mod**. f, in this case, is thus a constant morphism assuming a specific value $f(x) \in M$.[20]

Furthermore, all of the applications seen so far, are based on the category of modules **Mod** or the topos of modules $\mathbf{Mod}^{@}$. Starting in 2002, Mazzola's theory started being used to describe other constructs, for instance the ones of musical performance, using the topoi on the categories of *Fields*, *Graphs*, etc.[21] The theories presented in Section 3.3.2 are another extension, using the categories of *Topologies* and *Graphs*.

Again, all the theories described in this section are theories of processes. They model how the spaces of musical facts can be created, how facts can be defined in them, and most importantly how transformations can be defined that map facts into other facts. In the latter theory, we can easily define transformations of transformations, thus processes of processes, and so on. What these theories do not account for, as seen above, are the way these transformations are executed. Neither morphisms nor transformations and operations have a spacial existence comparable to the objects themselves. They are discrete entities that define a beginning and an end and nothing more, which is why the term *gesture* seems inappropriate. The next section is concerned with theories that deal with this problem.

3.3 Gestures in Music Theory

As seen in Section 2.2, the term gesture is constantly used in the discourse about music, with respect to physical gestures, but also in a way detached from physicality. Nevertheless, even when used metaphorically or as a mental construct, gestures always

[20]For details, see Mazzola, *The Topos of Music. Geometric Logic of Concept, Theory, and Performance*.
[21]Ibid., p. 726.

keep a link to the human body and are products of embodied experience. In this section, we briefly survey a few examples of gestural theories in music, with a focus on Mazzola's theory of gestures, which was an inspiration for the work of this thesis.

3.3.1 Gesture Theories

Many theorists use the word gesture simply to describe a short and characteristic musical gestalt such as a melody, a motive, or a brief harmonic motion.[22] Some scholars employ the term slightly more specifically and have attempted to define musical gestures in a more systematic way.

For instance, Robert Hatten defines a musical gesture broadly as seen in Section 2.2, as a "significant energetic shaping through time".[23] He sees them as temporal gestalts that are affectively loaded and that musicians can learn as sensorimotor schematas and reapply to other contexts. In this respect, Hatten often considers gestures as closely related to classical topics and often identifies one with the other. However, as opposed to topics which can refer to larger cultural context or styles, such as in *hunt*, *pastorale*, or *learned style*, Hatten's gestures carry more basic affective meaning or behavioral schemes. For example, there are *rhetorical* gestures such as the cadential 6/4 that marks the break for a cadenza in a concerto, or *spontaneous* gestures that avoid musical formulae and express spontaneous movement, or *dialogical* gestures that emerge from a musical conversation.[24] For Hatten, gestures can become motivic or thematic when used more consistently.

Manfred Clynes was also among the scholars who criticized the lack of gesturality in Western musical notation, as did Adorno (see 2.2). Clynes argues for the existence

[22]Oded Ben-Tal. "Characterising musical gestures". In: *Musicae Scientiae* 16.3 (2012), pp. 247–61.

[23]Hatten, *Interpreting Musical Gestures, Topics, and Tropes*, p. 95.

[24]Robert Hatten. "A theory of musical gestures and its application to Beethoven and Schubert". In: *Music and Gesture*. Ed. by A. Gritten and E. King. Aldershot: Ashgate, 2006, p. 6.

of so-called essentic forms, biologically programmed carriers of emotional semantics. In music, these essentic forms are characterized by agogic and dynamic performance structures, the most basic of which is the four-beat pattern. Clynes found that the way this pattern is realized by performers when playing music of different composers varies greatly and that the music of each composer has a prototypical pulse associated with it.[25] Following this, Clynes found gestural aspects of performance that are passed on aurally that would be impossible to be captured by Western notation.

Oded Ben-Tal recently distinguished *expressive unit gestures* from figures and motives.[26] The former happens in the foreground of the music, its salient features are preserved when repeated, and always needs to be complete. In contrast, figures occur in the background, are repeated regularly with slight variations, and are typically incomplete. Finally, motives are a foreground phenomenon, are complete but open to extension, and are frequently developed. With this distinction, Ben-Tal considered expressive unit gestures as the auditory analogue of physical gestures. He then suggested the development of a descriptive system based on the Hamburg Sign Language Notation System, which characterizes musical gestures based on attributes such as stress patterns, dynamics, acceleration/deceleration, pitch contours, and register.

3.3.2 Gesture Theory as an Extension of Transformational Theory

In the introduction to this thesis, we discussed Lewin's notion of gesture, which strongly differs from the notions of other scholars discussed above. The term gesture

[25]Later on, Clynes led a study where compositions of four composers were played back in each of the four composer's pulses. All groups of subjects significantly preferred the ones in the original pulse associated with the composer, increasingly with a higher degree of musical training. Manfred Clynes. "Microstructural Musical Linguistics: composer's pulses are liked best by the best musicians". In: *COGNITION, International Journal of Cognitive Science* 55 (1995), pp. 269–310.

[26]Ben-Tal, "Characterising musical gestures", p. 254.

does not characterize the musical gestalt itself, but the the process of one gestalt becoming another. It does not take a gesture as a fact but focuses on gesture being motion within the gestalt. For instance, the movement perceived in what traditional scholars consider a gesture is in fact a movement from note to note, or object to object, and not just there as a given relationship. Lewin has a more phenomenological way of looking at this.

We also saw that transformational theory was partly criticized for its incapability to formalize the gestures Lewin mentioned in connection with his theory. Several scholars have come up with theories that model continuous space and consider transformations within this space.[27] However, the first theory that extends transformational theory with its graphs and networks and models the gestures themselves rather than a continuous space is Mazzola's gesture theory.[28] It models gestures and describes how processes are related to them by using two constructs. The *skeleton* is a directed graph Γ, just as Lewin's, and thus a process. A *gesture* is a morphism $g : \Gamma \to \overrightarrow{X}$, where X is the gesture's topological *space* and $\overrightarrow{X}$ is the set of all continuous curves in X, i.e. all $f : I \to X$ with $I = [0, 1] \subset \mathbb{R}$, the closed interval $0 \leq r \leq 1$. The so-called *body* of the gesture is its image, i.e. all curves in $\overrightarrow{X}$ that are reached by g. Figure 3.2 shows a gesture in $X = \mathbb{R}^3$ with its skeleton Γ.

Now what can we do with this construction? It all depends on what space we select as X. For instance, if we wish to model hand gestures, we might decide to take a four-dimensional space consisting of three spatial gestures and time. On the other hand, in order to model musical gestures on a score, we might be content with a simple two-dimensional space that corresponds to the sheet of paper, one of the dimensions being

[27] One example is Tymoczko's geometrical model using orbifolds. Dmitri Tymoczko. *A Geometry of Music*. New York: Oxford University Press, 2011.

[28] First published in Mazzola and Andreatta, "Formulas, Diagrams, and Gestures in Music". More broadly described in Mazzola and Cherlin, *Flow, Gesture and Spaces in Free Jazz. Towards a Theory of Collaboration*, p. 81.

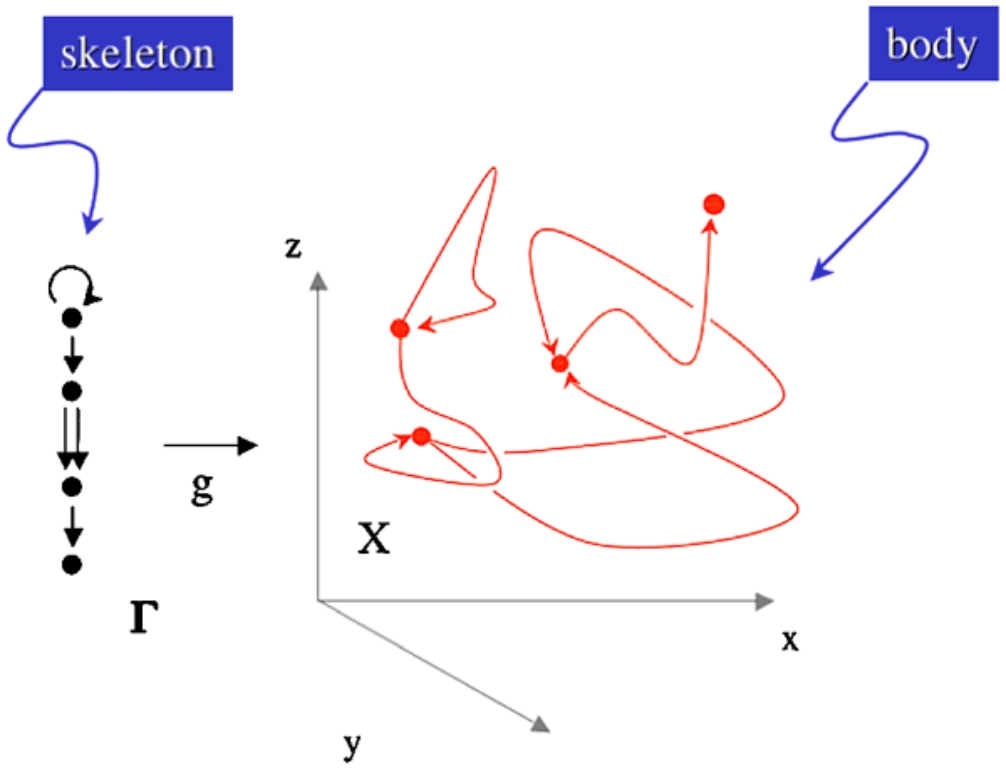


Figure 3.2: A sample gesture with skeleton $\Gamma = (\{0,1,2,3\}, \{(0,0),(0,1),(1,2),(1,2),(2,3)\})$ and space $X = \mathbb{R}^3$.

time, the other pitch. However, if we consider musical objects as higher-dimensional, such as in the example in the previous section. It is important to note that X does not have to correspond to an object space, but can be independent. For instance, in our software, everything can be controlled using mouse gestures, which are simply based on a two-dimensional space (enhanced by a few discrete spaces defined by the buttons). The object space, however, can be arbitrarily defined by the user.

How do gestures relate to the formalisms presented in the previous section? As mentioned above, the gesture's skeleton Γ is a digraph analogous to Lewin's graphs and networks and analogous to the diagrams common in category theory. Its spacial configuration does not show more or less than transformation networks and diagrams of morphisms. In the above constructions, the space in which the digraphs are imag-

ined does not have anything to do with the space where the musical objects are located and transformed. It is merely a logical space that shows causal relationships. Even the way in which transformations are attached to the graphs in processual theories is analogous to the way gestures are attached to the graph in gesture theory. The difference is that while in processual theories, the transformations occur in a discrete way, here they are curves in a space. To get from $x_1 \in X$ to $x_2 \in X$, an infinite amount of paths (curves) can be traveled. In other words, for a certain skeleton Γ we can find an infinite number of realizations in $\overrightarrow{X}$, just as we can find an infinite number of gestures fulfilling a certain action, as described in Section 2.2. Similarly, for every gesture, we may find an infinite number of segmentations that lead to new definitions of processes. For instance, for every arrow $a \in \Gamma$, we may introduce an intermediary state by selecting a point $x \in X$ with $x \in g(a)$ to obtain two arrows a_1, a_2 with $f(1)(g(a_1)) = x$ and $f(0)(g(a_2)) = x$, which divides a partial gesture in g into two sub-gestures.[29] Ultimately, we can still take the same spaces as above, as long as the are of topological nature.[30]

With this definition of gesture we can also find morphisms between gestures. For $g : \Gamma \to \overrightarrow{X}$ and $g : \Delta \to \overrightarrow{Y}$ a *morphism between gestures* is a pair (u, v) with $u : \Gamma \to \Delta$ and $v : X \to Y$ such that $h \circ u = \overrightarrow{v} \circ g$. This allows us to for instance map the gestures of a musician onto the gestures we perceive in the resulting music. Or in a more practical application where we map the parameters of a gestural computer interface onto musical parameters as it is done for the various interfaces supported by *BigBang*, which I will describe later on.

On a higher level, we can also define hierarchically related gestures, i.e. gestures

[29]This can in fact be done in *BigBang*, as will be shown in Section 7.3.4.

[30]It is more likely to find topological analogues to module spaces. However, most spaces in *GMIT* can be converted into module spaces and subsequently into topological spaces. This will be discussed in a forthcoming article.

of gestures. We can formalize this by defining a new topological space $\Gamma\overrightarrow{@}X$ of all gestures with the same skeleton Γ and the same space X. If we now define a morphism $h : E \to \Gamma\overrightarrow{@}X$, we obtain a so-called *hypergesture* with skeleton E, a gesture of gestures.

In recent years, the importance of gestures and continuous spaces has been increasingly felt, perhaps initiated by Lewin's visionary ideas, and scholars have sought to find formal descriptions for them. However, what distinguishes Mazzola's gesture theory from other attempts to formalize continuous musical and non-musical movements, is that it integrates both process graphs, and movements in continuous space. Some other theorists have uniquely dealt with the latter, a formalization of geometrical space, thereby losing what holds transformational theory together, the graphs with their causal implications.[31] Other scholars have attempted to bring movements into graph or network space, such as Roeder with his animations. Mazzola's solution keeps the gestures in the space of the musical objects, but connects them logically with the graphs needed to theorize about specific instances or subjective perceptions, just as Lewin envisioned it. The paradigm of gestures is of great significance for many domains and has been applied practically years before it was formalized by Mazzola. In the next chapter, we will encounter such applications and sometimes discuss their connections to gesture theory.

[31]Such as for instance Tymoczko, *A Geometry of Music*.

Chapter 4

Facts, Processes, and Gestures in Composition and Improvisation

In this chapter we are concerned with more practical aspects of the dimension of embodiment. After a brief look at the notion of composition and improvisation underlying this thesis, we examine several examples of how facts, processes, and gestures are perceived by practical musicians, with a focus on computer-assisted music-making. The final sections of this chapter serve as an introduction to and critique of *Rubato Composer*, the software for which *BigBang* was created.

4.1 Some Thoughts on Composition and Improvisation

Since one of the main uses of the software developed in the context of this thesis is active music making in the form of composition and/or improvisation, it is necessary to consider some of their aspects and their relationship. In Chapter 2, we have already briefly considered the nature of musical works and how their process of creating can

be formalized. However, we have not yet considered the temporal and contextual aspects of the creative process, which are deemed important by many musicians.

Any act of creating music can be situated somewhere on a line in between the two extremes of entirely spontaneous free improvisation and strictly preconceived algorithmic composition. The former happens entirely in the moment and ideally involves no pre-planned structures. The latter, in turn, typically happens in an iterative process outside of musical time and the musical result is entirely predictable. However, these two extremes hardly ever happen in reality. Most improvising musicians constantly and systematically work on their instrumental skills and their repertoire of musical ideas for decades, and could therefore be seen as composing their improvisations. On the other hand, computer musicians in the process of creating a system architecture for algorithmic composition almost always rely on spontaneous decisions and solutions and thus often compose their seemingly rigid systems in an improvised way. Furthermore, the compositional results of such a system are often selected spontaneously based on aesthetic notions or influenced by having access to certain musical parameters. From this we can infer that all music making procedures contain aspects of both composition and improvisation, just as every other activity in our lives include planned and spontaneous factors.

Several scholars have brought to attention that the difference between composition and improvisation is often overestimated.[1] Similarly, we defined any kind of music

[1]For instance in Philip Alperson. "On musical improvisation". In: *Journal of Aesthetics and Art Criticism* 43.1 (1984), pp. 17–29, p. 17. Some scholars make a similar case for musical performance, which again requires both "composed" (meticulously practiced) and improvised (spontaneous adaptation to instrument, setting, atmosphere, mood, etc.) factors. Carol S Gould and Kenneth Keaton. "The essential role of improvisation in musical performance". In: *Journal of Aesthetics and Art Criticism* 58.2 (2000), pp. 143–8 However, other scholars see improvisation more rigidly and merely consider changing structural aspects of the music as improvisation, for instance James O. Young and Carl Matheson. "The metaphysics of jazz". In: *Journal of Aesthetics and Art Criticism* 58.2 (2000), pp. 125–33. However, since we may consider expressive parameters as structural aspects of a performance, our notion here is that improvisation consist in any spontaneous activity.

making as the sum of improvisation and composition.[2] This is based on our view that any spontaneous musical activity can be called improvisation. In this thesis we follow this notion by claiming that a successful context for music making must both allow for rigorous planning and leave room for spontaneity. Composition can bring about results of a refinement that would not be conceivable in improvisation. On the other hand, the liveliness and ingenuity of improvisation is often irreproducible in a compositional context.

This is especially significant when using a computer to compose or improvise. The distances we discussed in the introductory chapter that often occur in connection with formalisms or technology may hinder composers to be both spontaneous and planned enough. The software subject to this thesis allows for both, by enabling intuitive controls of formal matters that encourage spontaneity, and by precisely tracking the musician's process and again allowing spontaneous interaction with it. Our software should allow composers to improvise until they find what they are looking for, and improvisers to pre-compose certain parts of their improvisation. One way to achieve this is the incorporation of the three levels of embodiment, described above. Before I describe how this works, it will be helpful to look at some other software, discuss their relationship to these three levels, and investigate the relationship between gestures and of intuitive control. First, I will briefly summarize Mazzola's ideas about free jazz, which are a worthy starting point for the following discussions.

[2]Sounding music = composition + improvisation. Guerino Mazzola, Joomi Park, and Florian Thalmann. *Musical Creativity – Strategies and Tools in Composition and Improvisation.* Heidelberg et al.: Springer Series Computational Music Science, 2011, p. 234f.

4.2 Gestures in Improvised Music

In their book on free jazz, Mazzola and Cherlin characterize improvised music by the art of collaboration, which based on three fundamental concepts: a collaborative space, gestural creativity and communication, and group flow.[3] There are two types of *collaborative spaces*: closed and open ones.[4] The former is given a priori and can be seen as a constraint to the performers, such as tonalities, scales, or fixed formal schemes, and the latter can be shaped, deformed, manipulated, or created by a single musician or the group. Any of the former may be opened up if it is physically possible and the group agrees on it. Mazzola especially often refers to the activity of improvisation consisting the shaping of time, which then becomes an open space. Earlier on, the authors draw an analogy with Bill Wulf's notion of a collaboratory as a center without walls, which can be interpreted in any physical, mental, or psychological way.[5]

The authors understand *gestures* as an intuitive way of communication apart from semiotic implications in the sense of the French school as described in Section 2.2 of this thesis.[6] Just as scholars from different fields or speakers of different languages increase their use of gestures when communicating with each other, musicians collaborating in free improvisation, who often have an entirely different musical background, understand each other on a more intuitive level and use musical gestures apart from theoretical constructs and avoiding any extramusical implications. Later on, the authors use the mathematical constructs defined in Section 3.3.2 of this thesis to describe

[3]Mazzola and Cherlin, *Flow, Gesture and Spaces in Free Jazz. Towards a Theory of Collaboration*, p. 34.

[4]Ibid., p. 42.

[5]ibid., p. 34. Mazzola's concept of musical creativity is closely related to this notion: identifying the walls of a concept, opening them, and evaluating the newly created space. Mazzola, Park, and Thalmann, *Musical Creativity – Strategies and Tools in Composition and Improvisation*, p. 17.

[6]Mazzola and Cherlin, *Flow, Gesture and Spaces in Free Jazz. Towards a Theory of Collaboration*, p. 65.

gestural interaction between different musicians.[7] Assumed that each musician plays a different instrument and has a different musical background (their topological space), the adaptation and deformation of one musician's (hyper-)gesture to another's context may be described as a morphism between gestures of different topological morphisms, so-called "throw morphisms", followed by a deformation morphism within the new topological space.

Finally, *flow* is a concept defined by Mihály Csíkszentmihályi and described as a state of "being completely involved in an activity for its own sake. The ego falls away. Time flies. Every action, movement, and thought follows inevitably from the previous one, like playing jazz. Your whole being is involved, and you're using your skills to the utmost."[8] More formally, Csíkszentmihályi defines flow as a state where a subject is challenged exactly to a degree appropriate to their skill level, and neither over- nor under-challenged. In order to enter the state of flow, one must have a clear goal, there must be immediate feedback during the process on how well one is doing, and one must have confidence in oneself at any stage of the process.[9] In the state of flow, one in concentrated and motivated to such a degree that one loses one's sense of self and forgets everything happening outside the task. Keith Sawyer expanded on this concept by defining *group flow*, where based on similar conditions as in Csíkszentmihályi, a collaborative group can develop a distributed identity where their egos blend, when all members listen and participate appropriately.[10]

It is important to note that the first two concepts, space and gesture, and the

[7]Mazzola and Cherlin, *Flow, Gesture and Spaces in Free Jazz. Towards a Theory of Collaboration*, p. 92.

[8]Interview with Csíkszentmihályi in John Geirland. "Go With The Flow". In: *Wired magazine* 4.09 (Sept. 1996).

[9]Mihály Csíkszentmihályi. *Flow: The Psychology of Optimal Experience*. New York: Harper and Row, 1990.

[10]R. Keith Sawyer. *Group Creativity: Music, Theater, Collaboration*. Mahwah, NJ: Lawrence Erlbaum Associates, Publishers, 2003.

implied openness and intuitiveness, encourage the occurrence of flow. Similarly, our concept of musical creativity depends on opening up a limited conceptual space, which is especially facilitated by a gestural way of thinking.[11] Creativity and flow are directly related, the former consisting of a temporary ease and constant capability of finding of creative solutions to emergent problems. These are also the findings of computer musicians, the writings of whom are the topic of the next section.

4.3 Embodiment and Interactive Composition Systems

Both the previous chapter and the previous section gave examples of how gestures and processes can be found or incorporated in music, be it composed or improvised. Since the focus of this thesis is how the levels of embodiment can be applied to computer-assisted composition and improvisation, in this section we will take a closer look the writings of computer musicians and some example systems that use processual and gestural paradigms. I will move from facts to gestures and discuss recent theoretical and practical achievements.

From their early days, computers were used to create music in ways that had been inconceivable before. While initially each musical work was typically composed using a setup specific to the piece, the 1970s saw an advent of interactive composition systems that were designed for a variety of composers to create multiple works, such as Chadabe's *Play* system[12] or Iannis Xenakis's *UPIC*.[13] Such systems are typically

[11] Mazzola, Park, and Thalmann, *Musical Creativity – Strategies and Tools in Composition and Improvisation*.

[12] Joel Chadabe. "Interactive Composing: An Overview". In: *Computer Music Journal* 8.1 (1984), pp. 22–7, p. 22.

[13] G. Marino, M.-H. Serra, and J.-M. Raczinski. "The UPIC System: Origins and Innovations". In: *Perspectives of New Music* 31.1 (1993), pp. 258–69.

referred to as *computer-assisted composition* systems and are distinguished from algorithmic composition systems. The former are designed to be more interactive than the latter and even though they may use complex algorithms to create musical results, users do usually not have to be computer experts or know how to program, while they can still have a great deal of control over the system.

4.3.1 Thinking and Making Facts or Objects

When composers or improvisers use computer systems, compared to using traditional instruments and staff notation, they often learn to rethink what musical structures and objects are. Computers and programming languages set almost no limits as to what the constructs are that we can make music with. Even though there are many software products that allow composers to write in staff notation or simulate the sound of orchestral instruments, there is an immense variety of differing approaches each of which has its advantages and disadvantages. What are the facts or objects that users of such systems can work with, how are they visualized, and how can they be created?

The objects that interactive composition systems allow their users to create and manipulate can best be classified in respect to the ontological topography discussed in Chapter 2. Analogous to the three levels of reality, we find mainly physical and symbolic objects, and arguably sometimes also psychological ones. Among *physical objects* we find digital representations of real-world physical structures such as sampled sound waves, physical models of instruments and their extensions, synthetic sound structures created with additive synthesis or frequency modulations, or processing objects such as filters, delays, reverb, etc. Often, these objects can be built up in a modular way from even more basic objects, similar to the way that real-world synthesizers can be programmed. *Symbolic objects* do not directly represent sounds

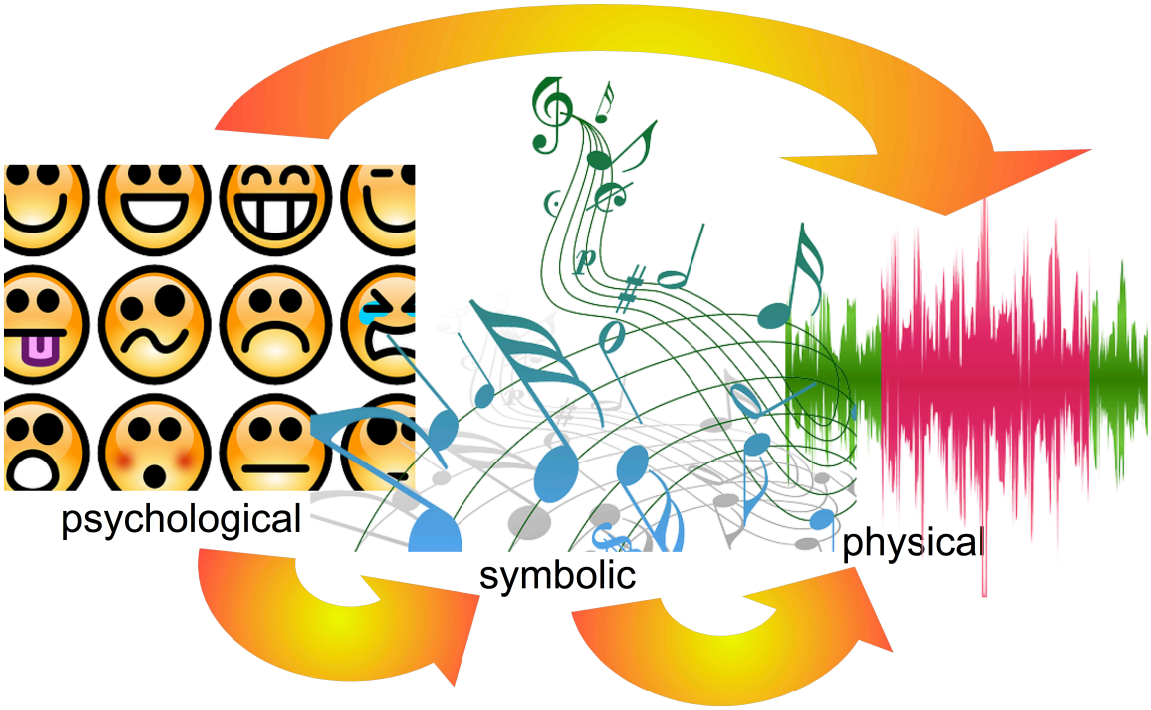


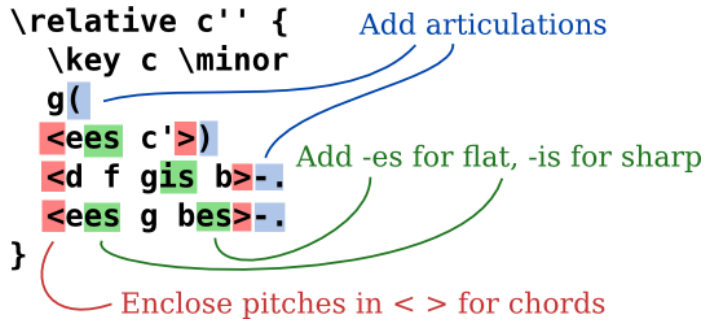
Figure 4.1: The types of facts in computer-assisted composition and how they are typically converted into each other.

and typically have to be converted into physical objects in order to become audible. They include score notation symbols, mathematical structures, MIDI objects, etc. Finally, there are some rare examples of *psychological objects* such as emotive states that again need to be converted into symbolic or physical objects in order to become audible.[14] Figure 4.1 depicts the overall conversion scheme.
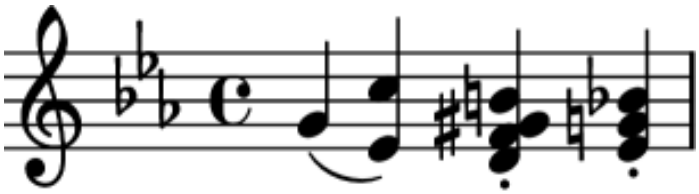
The ways such objects are defined and manipulated vary greatly from system to system. *Code-based* systems use notations based on programming languages. In LilyPond,[15] for instance, which is used to notate the score examples in this thesis as well

[14]Examples are Dan Wu et al. "Music composition from the brain signal: representing the mental state by music". In: *Computational intelligence and neuroscience* (2010) or uses of Manfred Clyne's sentograph, as described in Alf Gabrielsson and Erik Lindström. "Emotional expression in synthesizer and sentograph performance". In: *Psychomusicology: Music, Mind & Brain* 14.1 (1995), pp. 94–116.

[15]Han-Wen Nienhuys and Jan Nieuwenhuizen. "LilyPond, a system for automated music engraving". In: *Proceedings of the XIV Colloquium on Musical Informatics*. Firenze: CIM, 2003.



(a)



(b)

Figure 4.2: An example from LilyPond, (a) as code and (b) as staff notation (source: http://lilypond.org/text-input.html).

as in a module for *Rubato Composer* developed while writing this thesis (*LilyPondOut* rubette), symbolic objects are notated as groups of letters and numbers standing for pitch and register, and grouped into chords, phrases, and so on, using brackets and parentheses (see Figure 4.2).[16] Other code-based systems specialize on physical objects, such as SuperCollider or CSound, and because of the efficiency and versatility of programming code, users can create highly complex musical constructions with just a few lines of code.

Visual systems with a graphical user interface usually find a wider base of users for they are often accessible in a more intuitive way than code-based systems, but they often have limitations because of their visuality leads to specific musical constraints. For symbolic objects, the most widely used systems are *staff notation programs* such as Sibelius or Finale, which allow a great variety of musical objects to be drawn

[16]For an in-depth comparison of symbolic code-based systems, see Eleanor Selfridge-Field. *Beyond MIDI: the handbook of musical codes*. Cambridge, MA: MIT Press, 1997.

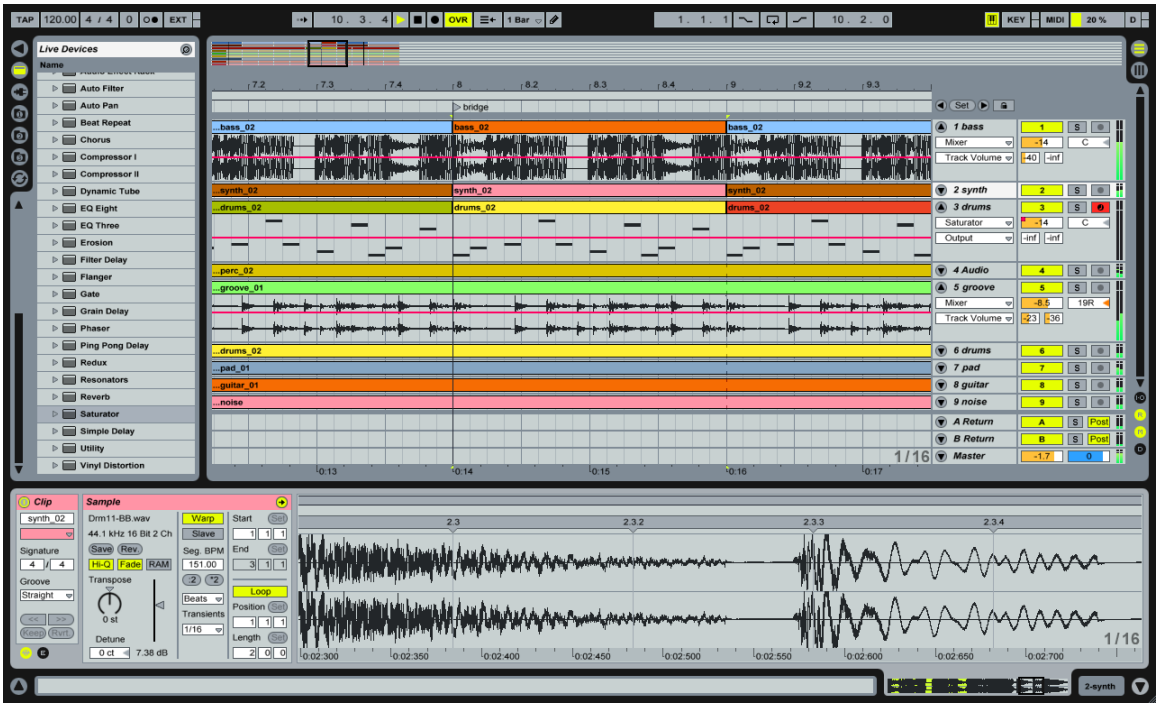


Figure 4.3: Ableton's sequencer *Live* with a few audio (sound waves) and MIDI (piano roll) tracks.

on staves. Even more commonly used than notation systems are *sequencer softwares*, which combine symbolic and physical objects and are mainly used for music recording and production. They typically present musical content as a number of tracks laid out as horizontal rows, the horizontal dimension representing time. There are usually two types of tracks, holding either symbolic MIDI or physical audio data. As an alternative to staff notation, MIDI data is often displayed in so-called piano roll notation, where notes are represented as rectangles of different dimensions on a two-dimensional plane, inspired by the perforated paper rolls of nineteenth-century player pianos (see Figure 4.3).

A more simplified and unified way of visualizing musical objects was implemented in Iannis Xenakis's *UPIC* system (depicted in Figure 4.4. It was based on earlier visual composition systems such as Max Mathew's *Graphic 1*. The goal of *UPIC* was

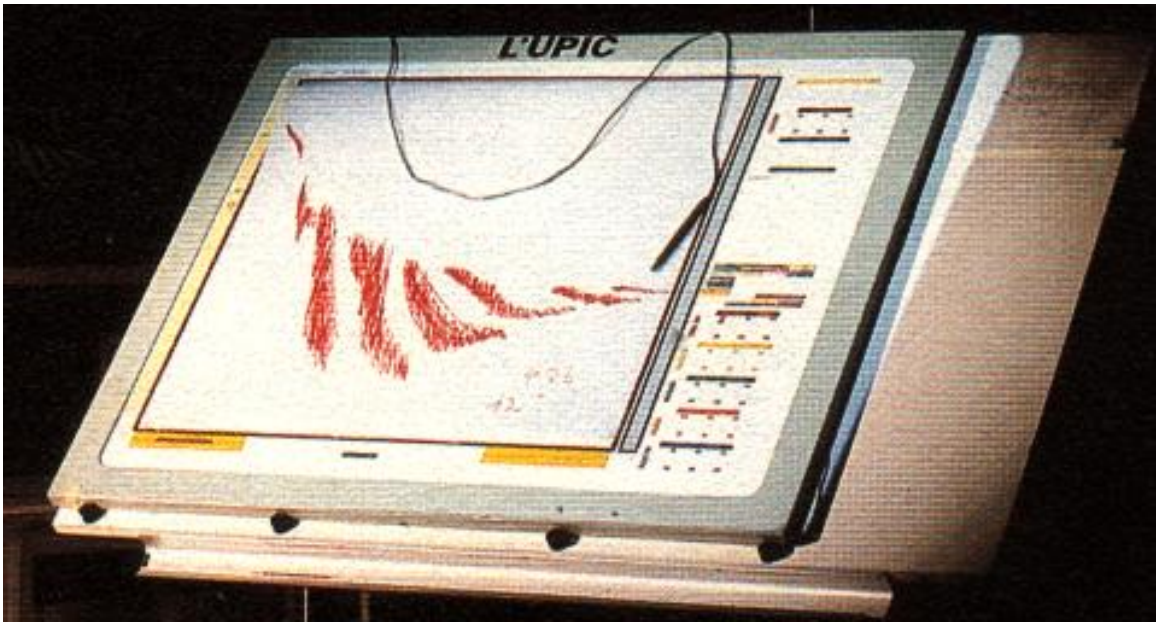


Figure 4.4: Xenakis's *UPIC* system.

to abolish the hierarchy between sound design and musical architecture, and to allow composers to "work in a systematic way on various levels at the same time."[17] Xenakis thought that the musical effects he wanted to compose with were too complex to be specified with traditional staff notation. *UPIC* enabled users to draw sound waves and organize them into a musical structure, all by interaction through a graphical tablet. It allowed composers to think on a more abstract level, away from formulas or program code, and provided intuitive and immediate exchange between thought and ear.[18] Frequency, amplitude, timbre, and time could all be controlled by the same objects, drawn curves, and positioned in a timeline to create a musical form. Several later systems were strongly influenced by *UPIC* and adopted both its graphical interface and the abolition of object hierarchies, for instance Hyperscore, HighC, IanniX, Sonos, or MetaSynth.

[17]H. Lohner and I. Xenakis. "Interview with Iannis Xenakis". In: *Computer Music Journal* 10.4 (1986), pp. 50–5, p. 50.

[18]Marino, Serra, and Raczinski, "The UPIC System: Origins and Innovations".

These examples show that composition or improvisation systems are usually focused on a specific group of musical objects, letting users access them through a specific interface. Some systems allow the users to redefine the objects themselves. This is naturally the case with code-based systems that are often extended by their users who can design larger objects from smaller ones and even introduce new ones. However, in visual systems this is usually not the case. Even *UPIC*, despite its anti-hierarchical concept, limits its user to varying only the predefined musical parameters. Similarly, the ways in which objects can be manipulated are usually given by the system and not extensible.

In contrast, the software we work with in this thesis, *Rubato Composer*, builds on a more general definition of object types and lets users create their own arbitrary objects at runtime. Almost all the functionality applies to any object type. We will see this in the last section of this chapter.

4.3.2 Composition Systems and Processes

Several scholars have focused on modeling the creative process of composers with computers and created systems that automatically compose music in certain styles, while others have created systems that track composer's processes.[19] Few, however, have developed software products that track the composer's process and let them interact with it in a more dynamic way. Most software products use a plain undo/redo functionality, which is a simple instance of processes. The users have the opportunity to undo the last sequence of operations performed and redo them if they decide to. This corresponds to a one-dimensional sequential process graph, even if it is not visualized this way. Each node of the graph represents a specific state of the musical

[19]Examples are David Cope. *Computer Models of Musical Creativity.* Cambridge, MA: MIT Press, 2005, or David Collins. "A synthesis process model of creative thinking in music composition". In: *Psychology of Music* 33.2 (2005), pp. 193–216.

work containing the musical objects present at that time.

However, several composition and improvisation systems use processes in a different way. One possibility is that the operations available to the user become objects themselves and can be connected using connective lines reminiscent of electric cables, and often inspired by analog synthesizer interfaces. *Max/MSP* and *Pure Data (Pd)*, the prominent signal processing software systems, for instance, provides an immense variety of objects that perform certain actions such as generate an oscillator signal, add or multiply signals, take a MIDI input, provide visual controls, etc. The users can develop networks of such objects, combine them into higher level macro objects, and manipulate them using the various control mechanisms.[20] When using these products, users do in fact create process diagrams similar to graphs in transformational theory, however, with the major difference that they are the dual graphs to transformational networks. In other words, the nodes correspond to actions or transformations whereas the connecting lines, just as unidirectional as digraph arrows, correspond to the objects themselves (see Figure 4.5). *Rubato Composer*, introduced below, is based on precisely the same principle, as are for instance Open Music, or Reactable.

Some of these network-oriented systems, including *Pd* or *Max/MSP*, allow for streams of objects instead of single objects to be sent along the connective lines. This is crucial for real-time signal processing, since its objects are of continuous nature, and it is a necessary condition for gestural control, which will be discussed in the next section.

[20] Miller Puckette and David Zicarelli. "MAX - An interactive graphic programming environment". In: *Opcode Systems, Menlo Park, CA* (1990); Miller Puckette. "Pure Data: another integrated computer music environment". In: *Proceedings of the International Computer Music Conference*. 1996, pp. 37–41.

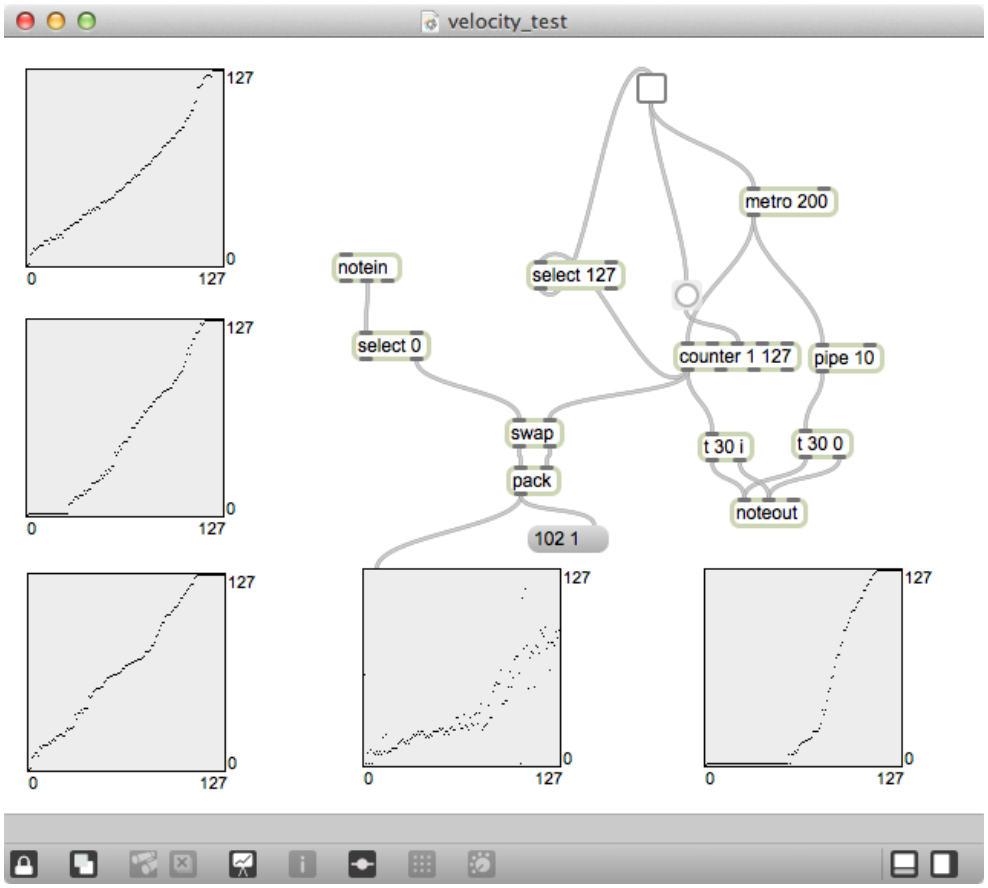


Figure 4.5: A Max/MSP patch I created to test the velocity calibration of a MIDI grand piano. Each visible object performs an action or transformation while the musical objects, MIDI notes, travel along the connective lines.

4.3.3 Gestural Interaction with Composition Systems

Types of Gestural Interaction

Already some of the early composition systems allowed for gestural interaction, which is often thought to consist in recognition of predefined hand gestures but which may include any continuous way of interaction, be it with spacial sensors or a regular computer mouse, or even more generally, any way in which human gestures find their way into the music. Chadabe, for instance, describes how with his *Play* system, a composer could first design an algorithmic composition process and then interact

with it gesturally while the music was playing back. Using either proximity sensors or keyboard controllers the composer could influence certain parameters of the resulting composition, such as tempo or timbre. *Play*'s two-step procedure is somewhat analogous to composing and performing, even if Chadabe considers both as compositional activities.[21] On the other hand, Xenakis's legendary *UPIC* system shows a very different way of interaction. By drawing shapes on a tablet, the composer could gesturally define the composition itself rather than its performance, and each performance of a composed piece would typically be the same. In comparison, the two types of gestural interaction may be categorized as *real-time* and *non-real-time*, based on the time of interaction being either synchronous or asynchronous with musical playback time. In the former type, composers virtually interact gesturally with the music itself, whereas in the latter type they compose their music gesturally and find these qualities again when they listen to their piece.

Another way in which gestural interactive systems may differ is in whether or not they are based on a *gesture recognition* system that identifies a discrete set of gestures, each of them of continuous nature. Systems using recognition are typically *operation-based*, which means that any member of an available set of operations can be triggered through a detectable gesture. There are examples of systems that allow users to define their own gestures by training the system, for instance IRCAM's *Gesture Follower*.[22] This way of interacting with a system has also gained significance in recent years, for instance with the emergence of multi-touch devices, which usually support a set of predefined simple gestural actions based on spacial metaphors, such as twist, pinch, or swipe, to transform objects on the screen or change the view. For a more thorough discussion of multi-touch gestures, refer to Section 8.1.2.

[21] Chadabe, "Interactive Composing: An Overview", p. 23.

[22] Frédéric Bevilacqua and Remy Muller. "A gesture follower for performing arts". In: *Proceedings of the International Gesture Workshop*. 2005.

A majority of the contributions to gestural interaction theories have focused on real-time non-operation-based systems. Here, however, we consider any interaction as gestural if the continuity of human gestures finds its way into the music in some recognizable fashion.

When is Interaction Gestural?

First, what are the optimal conditions of an interactive system in order to be considered gestural? Hunt and Kirk, and Wessel and Wright gather some of these conditions for real-time multi-parametric control systems.[23] The following list contains the most important ones in a reordered, regrouped, and expanded way:

- **Continuity.** Gestures should be executed in a continuous space and their musical counterparts should be equally continuous and thus equally spacial.
- **Immediate feedback.** Any gestural movement should lead to an immediately noticeable change in the music. Latency should be as low as possible.
- **Physicality.** The device through which the musician communicates should be operated through physical movements.[24]
- **Intuition.** The way of interaction can be learned easily and eventually be automated via motor-skills. Similar movements produce similar results. All movements produce understandable and more or less predictable results.

[23] Andy Hunt and Ross Kirk. "Mapping Strategies for Musical Performance". In: *Trends in Gestural Control of Music*. Ed. by M.M. Wanderley and M. Battier. Paris: Ircam - Centre Pompidou, 2000, p. 232. David Wessel and Matthew Wright. "Problems and Prospects for Intimate Musical Control of Computers". In: *Computer Music Journal* 26.3 (2002), pp. 11–22.

[24] Some scholars consider the acquisition of gestures through other parameters such as in Wanderley's indirect acquisition, where gestures are for instance acquired through analysis of sound.Marcelo M Wanderley. "Gestural Control of Music". In: *International Workshop Human Supervision and Control in Engineering and Music*. 2001, pp. 632–644, p.636-7 This, however, can only be considered as indirect gestural control. In fact, it is an example of analysis and gesturalization as defined in Section 2.3, where gestures are recreated from factual results. The procedures of formalizing and factualizing are missing in such a system.

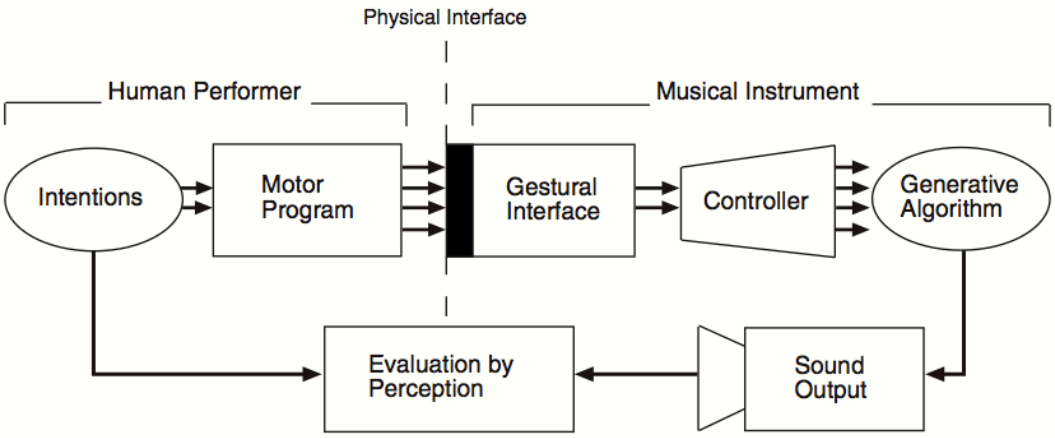


Figure 4.6: Wessel and Wright's two-way scheme of gestural interaction between humans and computers.

- **Virtuosity.** Interaction should allow for sufficient intimacy after practice so that competence and skill can increase unlimitedly over time.

- **Bidirectionality.** There should be no fixed order in the human-computer dialogue, but the human should be in control of the situation.

In short, the entire system should be built on a circular two-way interaction scheme where each action a composer performs leads the computer to generate music based on algorithms which is then immediately played back and, in turn, lets the composer react, and so on. Figure 4.6 shows Wessel's and Wright's schematization of this interaction scheme.[25] Wanderley further distinguishes primary and secondary feedback the former of which includes visual, auditory, and tactile feedback from the interface, and the latter feedback from the computer system.[26] A reversal of the definition might be more appropriate, since the second is usually what musicians are primarily interested in.

[25]Wessel and Wright, "Problems and Prospects for Intimate Musical Control of Computers", p. 12.

[26]Wanderley, "Gestural Control of Music", p. 635.

The above conditions assume a single long-term gesture being executed in a performative way, as typical for non-operation-based systems. However, gestural control may as well consist in many subsequently executed gestures, each of them interpreted in different ways and mapped to different operations, even if all communicated through the same controller. For instance on a multi-touch device, drawing and pinching gestures typically lead to different results. Are there any additional requirements for such gestural interfaces? In an earlier paper, we defined our own gestural interaction concept.[27] In addition to the above conditions, operation-based systems need to fulfill some needs concerning the operations themselves. The following list is an extension of our previously published one:

- **Atomicity.** Each operation should be executed with a minimal amount of gestures, ideally a single one, in order to be quickly and precisely applied in an improvisational way.

- **Variety.** The vocabulary of available gestures needs to contain maximally distinct gestures so that they can be recognized in an optimal way and so that the user can distinguish them easily.

- **Undoability.** If gestures can be applied easily and quickly there may be unwanted effects, where operations should be able to be undone and redone as easily as they were applied.

These additional conditions allow the musician to interact with the system seamlessly without having to focus too much on the interface itself. Wessel and Wright gathered several metaphors they used in their interfaces: drag-and-drop, scrubbing, dipping,

[27] Florian Thalmann and Guerino Mazzola. "The BigBang Rubette: Gestural Music Composition with Rubato Composer". In: *Proceedings of the International Computer Music Conference*. Belfast: International Computer Music Association, 2008, p. 3.

and catch-and-throw.[28] All of these are examples of operations that fulfill the above conditions. However, most of them assume the use of an interface where all gestural movements are considered independent. Today, with multi-touch interfaces, we are used to many more such metaphors, many of them based on more complex multi-finger gestures. There are also real-time operation-based systems such as *GIDE*[29] that use machine learning systems in order to create a user-defined vocabulary of gestures in order to trigger specific time-critical events and control the speed of their execution.

Gestures, Intuition, and Flow

Many scholars agree that gesturality, and all its characteristics compiled above, directly correlates with intuitiveness. Gesturality is most associated with our hands which are doubtlessly our most direct and natural way of interacting with the world. In fact, especially with our hands, we are often able to do things without needing to consciously think about them.[30] Simultaneously, gestures are often the most straightforward way in which we can understand things. And this is how intuition is usually defined, as the ability to understand something without the need for conscious reasoning. Intuitive interfaces are precisely the ones where users can forget about the fact that they are using an interface and where they can use prior innate, sensory-motor, cultural, or expert knowledge.[31] The better an interface can capture

[28] Wessel and Wright, "Problems and Prospects for Intimate Musical Control of Computers", p. 15-6.

[29] Bruno Zamborlin et al. "Fluid Gesture Interaction Design: Applications of Continuous Recognition for the Design of Modern Gestural Interfaces". In: *ACM Transactions on Interactive Intelligent Systems* 3.4 (2014).

[30] Analogous to Heidegger's example of the world as an extended craft shop, where the craftsman knows his stuff (Zeug) even if he is not able to explain it. Knowing how is prior to knowing that, meaning comes before the thing, and there is no distance between the Zeug and us.

[31] Katie Wilkie, Simon Holland, and Paul Mulholland. "What Can the Language of Musicians Tell Us about Music Interaction Design?" In: *Computer Music Journal* 34.4 (2010), pp. 34–49, p. 37, adopted from Hurtienne and Blessing.

the infinite variety of gestures we are capable of performing, and the faster *and* the better an interface can be learned, the more we can feel connected to it. Yet, how exactly do interfaces have to be configured so that they feel intuitive to us? Are there any general conditions in addition to the above ones that an interface should fulfill in order to become second nature?

One of the most important factors is the manner in which the gestural parameters are mapped to the musical parameters. Despite the potential possibilities of the myriads of available interfaces with gestural capabilities, most software products use them with so-called one-to-one mapping,[32] imitating slider or knob motion either directly or multidimensionally. Many interfaces that humans interact with in everyday life are highly multi-dimensional and contain complex mapping strategies, one-to-many, many-to-one, or many-to-many, rather than one-to-one. As Hunt and Kirk claim, humans even *expect* to encounter complex mappings[33] and they are far more ready to experiment with an interface if a simple gesture affects several parameters. Paradoxically, the more complex the mappings, the more simple the interface appears to be, and vice versa. A trumpet appears as nothing more than a pipe with three valves and a mouthpiece and there is no single control for any musical parameter. For instance, variation of pitch on a trumpet is a highly complex procedure that involves diaphragmatic tension, lip pressure, tongue position, valve position, and so on, and yet, we may instantly feel connected to it even if takes time to learn to play it. Then, on the contrary, the interface of an analog synthesizer with its many control knobs, sliders, switches, and patch cords seems highly complex to us even if each of its controls is typically mapped one-to-one to a single sonic parameter such as amplitude or frequency. A synthesizer interface appears daunting at first and a musician may

[32]Wessel and Wright, "Problems and Prospects for Intimate Musical Control of Computers", p. 11.

[33]Hunt and Kirk, "Mapping Strategies for Musical Performance", p. 234.

have to invest many years to intuitively understand the effect that each control has on the resulting sound.

Complex mappings can be achieved in a variety of ways. For instance, Hunt and Wanderley cite several examples of mapping gesture to sound using neural networks, instead of explicitly defining each relationship.[34] More explicit ways include interpolation between different sets of parameters, or an imitation of sculpting by means of gestural definition of geometrical shapes, the features of which are then in turn transformed into musical parameters, or the consideration of multi-level mapping.[35]

Most importantly, such mappings can only be intuitive if complying with certain cultural norms or personal preferences. Wilkie et al, for instance, researched ways in which simple conceptual models, image schemas such as UP-DOWN, CONTAINER, or SOURCE-PATH-GOAL, or conceptual metaphors can be used to determine the ways in which gestures are best mapped to musical parameters.[36] However, ultimately, flexibility in mapping may be most valuable and may add a lot of interest, since it allows users to experiment and adjust the system to the setting that seems most intuitive to them and that may lead to different and unique musical results.

Intuitiveness may also be increased when not every parameter of the musical system is directly accessible through the interface, but only through mappings with high-level generative parameters.[37] These parameters may be coupled together and even overlap to a certain degree, as in real-world interfaces. Hunt and Kirk suggest that multi-parametric or conceptual mapping allows the user to "think gesturally,

[34]Andy Hunt and Marcelo M. Wanderley. "Mapping performer parameters to synthesis engines". In: *Organised Sound* 7.2 (2002), pp. 97–108, p. 99.

[35]Ibid., p. 99-105.

[36]Wilkie, Holland, and Mulholland, "What Can the Language of Musicians Tell Us about Music Interaction Design?"

[37]Wessel and Wright, "Problems and Prospects for Intimate Musical Control of Computers", p. 12.

or to mentally rehearse sounds as shapes."[38] They define the so-called *performance mode*, an embodied type of computer performance, where musicians continuously control many parameters that are coupled together, with more than one conscious body parameter such as a limb or finger, and at the expense of physical energy.[39]

Finally, the more intuitive an interface ends up being, the more likely it is for the musicians to enter a state of flow,[40] as described above, being fully preoccupied with the momentary situation, just as in Lewin's being inside the music. Nevertheless, contrary to many scholars' characterizations, intuitiveness is not the sole prerequisite for flow to happen as musicians may easily get bored with an interface even if it is intuitive. As Csikszentmihalyi defined it, flow occurs when a subject is equally familiar with a situation but challenged just in the right way. So it is equally important for an interface to be challenging, complex enough to yield new musical situations, and configurable to a user's specific and unique ideas and needs.

Gestural User Interfaces and Controllers

Most graphical user interfaces of music software products widely used today are operated via a mouse or a multitouch screen and are based on skeuomorphs or contain skeuomorphic elements, i.e. imitations of real-world musical interfaces such as piano keyboards, synthesizer knobs, mixing desk sliders, tape tracks, piano rolls, etc. Even though many of their components could be described as rudimentarily gestural, for instance moving a slider with a mouse motion does in fact presuppose a gesture and immediate feedback is usually available, alternative interfaces may bear more musical potential. Playing a piano visible on the screen by clicking on its keys using a mouse

[38]Hunt and Kirk, "Mapping Strategies for Musical Performance", p. 255.
[39]Ibid., p. 233.
[40]Marc Leman. "Music, Gesture, and the Formation of Embodied Meaning". In: *Musical gestures : sound, movement, and meaning*. Ed. by Rolf Inge Godøy and Marc Leman. New York: Routledge, 2010, p. 139.

is a rather limited way of interacting with a computer.

This is why control devices, many of them specifically designed for musical purposes, have been around from the early days of computer-assisted composition. However, analogously to the graphical user interfaces, the most widespread physical interfaces are keyboard controllers and controllers with pads, knobs, and sliders are still the most widely used interfaces to computer music systems, to a large degree due to the limitations of the MIDI standard,[41] but also due to the universality of the piano keyboard and synthesizer controls and the needs of more traditional musicians. Many instrument-like controllers exist, doubtlessly because they rely on the skills of musicians trained on acoustic instruments. Wanderley divides musical controllers into three categories: instrument-like and instrument-inspired controllers, augmented instruments and hybrids with added sensors, and alternate controllers.[42] The first two categories include skeuomorphs and their extensions. Even if often classified this way, many of them are not fully gestural for they merely capture a discrete secondary effect of the musician's gestures rather than the continuous gestures themselves. A piano controller for instance, does not capture the pianist's gestures but discrete values calculated from the pressure exerted on the keys.[43] The fact that controllers are played using gestures does not mean that they are gestural and use continuous parameters.

In the context of this thesis we are most interested in fully gestural controllers and thus in Wanderley's last category, for their members bear the fullest potential of fulfilling the above conditions of gestural control.[44] Mulder distinguishes touch con-

[41] Wessel and Wright, "Problems and Prospects for Intimate Musical Control of Computers", p. 12.

[42] Wanderley, "Gestural Control of Music", p. 637-8. Eduardo Reck Miranda and Marcelo M Wanderley. *New digital musical instruments: control and interaction beyond the keyboard.* Vol. 21. Computer music and digital audio series. Middleton: A-R Editions, 2006, p. 20.

[43] Arguably the real instrument's hammer mechanisms do not do this either, but the pianist has the chance to interact with parts of the mechanism or the strings themselves.

[44] Levitin et al, for instance, discuss this potential especially in the light that musical controllers

trollers, expanded-range controllers that detect gestures in a limited field, and immersive controllers where any movement the musician makes has musical consequences.[45] The first two categories mainly classify two-dimensional and three-dimensional spacial interfaces often equipped with buttons, knobs, and so on, whereas the third one contains interfaces that are worn, such as gloves and bodysuits.

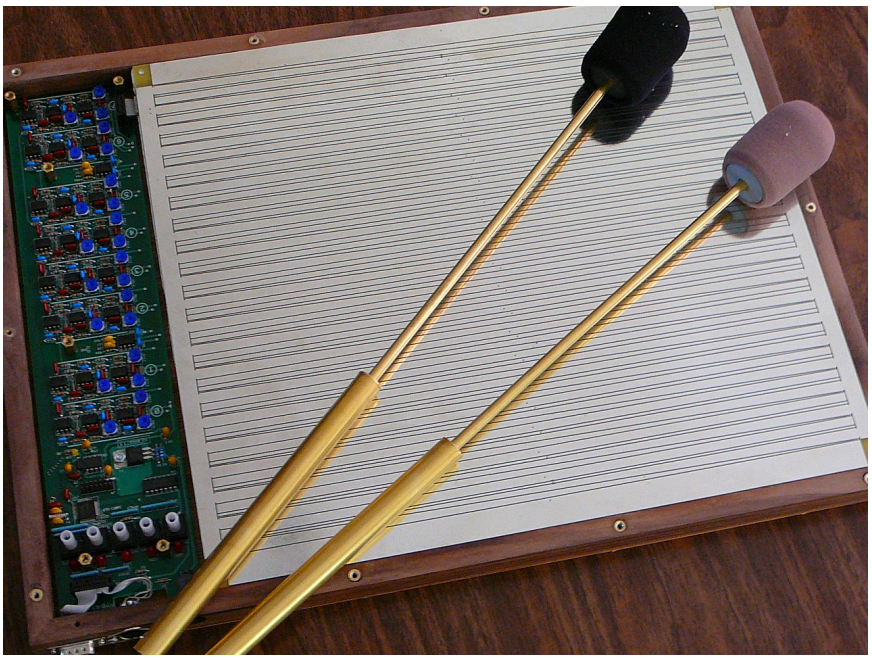
More important in terms of musical capabilities, however, is not the spacial dimensionality in which a controller detects continuous movement, but the dimensionality, degree of freedom, and interdependence of the various gestural parameters. For instance, both Max Mathew's Radio Batons and the Leap Motion device (both shown in Figure 4.7) are expanded-range controllers based on $\mathbb{R}^3$, but when both are played with two hands the former traces the position of two batons in this space, whereas the latter traces position *and* direction of palms and fingers of both hands. This corresponds to $2*2*3*6 = 72$, as opposed to $2*3 = 6$ continuous parameters.[46] On the other hand, we have to consider the fact that the finger of one hand and its palm only allow for so much independent motion, compared to two highly independent batons. I thus suggest to categorize gestural controllers with a triple (d, p, i) based on the dimensionality d of their topological space, the number of points p detected in this space when played by one person, and the number of entirely independent parameters i. Table 4.1 shows a few examples of gestural controllers and their categories.

This table shows the great variety and potential of different gestural controllers. Even the computer mouse has the potential to control gesturally, if its movements

do not have to include the sound-producing device anymore, as opposed to musical instruments, and may be designed entirely independently. Daniel J. Levitin, Stephen McAdams, and Robert L. Adams. "Control parameters for musical instruments: a foundation for new mappings of gesture to sound". In: *Organised Sound* 7.2 (2002), pp. 171–89.

[45] Axel G. E. Mulder. "Towards a choice of gestural constraints for instrumental performers". In: *Trends in Gestural Control of Music*. Ed. by M. M. Wanderley and M. Battier. Ircam - Centre Pompidou, 2000, p. 319-27.

[46] 2 hands, 2 for positions and vectors, 3 dimensions, 6 for fingers and hand, as opposed to 2 hands, 3 dimensions.



(a)



(b)

Figure 4.7: A juxtaposition of (a) Max Mathews's *Radio Batons*, and (b) *Leap Motion*.

controller	category
Ondes Martenot	$(1, 1, 1)$
Theremin	$(1, 2, 2)$
Computer mouse	$(2, 1, 1)$
Multi-touch surface	$(2, 10, 2)$
Radio Batons	$(3, 2, 2)$
Leap Motion	$(3, 24, 2)$

Table 4.1: Some examples of controllers with gestural capabilities.

are mapped in a continuous way. The ways we control our computers using the mouse, however, is not gestural per se. The path we travel from one click to the next most often has no influence on the functionality we trigger by clicking. Ultimately, gestural control does not solely depend on the controller, but is much rather a way of interaction with the system, as seen in the previous sections.

4.4 Rubato Composer

Let us now turn to the system that *BigBang*, the software implemented for this thesis, is built for: *Rubato Composer*. First, a brief overview of its precedents will be helpful, since some of them were an inspiration for *BigBang*, along with some of the systems discussed so far. Afterwards, I will describe the two main components of *Rubato Composer*, the underlying framework based on mathematical music theory as well as the graphical user interface.

4.4.1 Brief History

The earliest system in the genealogy of *Rubato Composer* was the computer with the memorable name $\mathbb{M}(2, \mathbb{Z}) \setminus \mathbb{Z}^2$-*O-Scope* developed by Mazzola in the early 1980s and shown at the *Salzburger Musikgespräche* in 1984 (Figure 4.8). As its name suggests, it provided a two-dimensional surface for $Onset \times Pitch$ on which users could perform



Figure 4.8: Mazzola demonstrating his $\mathbb{M}(2,\mathbb{Z})\setminus\mathbb{Z}^2$-*O-Scope* to Herbert von Karajan and an audience at the *Salzburger Musikgespräche* in 1984.

geometrical transformations of musical objects that could also be higher-dimensional.

The success of the system[47] led to the development of the commercial Atari ST software *Presto* (Figure 4.9), which appeared in 1989 and featured an elaborate graphical user interface and musical objects in a four-dimensional space $\mathbb{Z}_{71}^4$ ($Onset \times Pitch \times Loudness \times Duration$). In addition to the geometrical operations in all four dimensions there were other mathematical operations that could be performed. Its so-called *OrnaMagic* module (Figure 4.10) could generate so-called ornaments, regular grid structures created by translating a motive repeatedly[48] as well

[47] According to Mazzola, Herbert von Karajan wished to be left alone with the $\mathbb{M}(2,\mathbb{Z})\setminus\mathbb{Z}^2$-*O-Scope* for a night.

[48] Recently, these structures were generalized for any transformations and termed wallpaper and

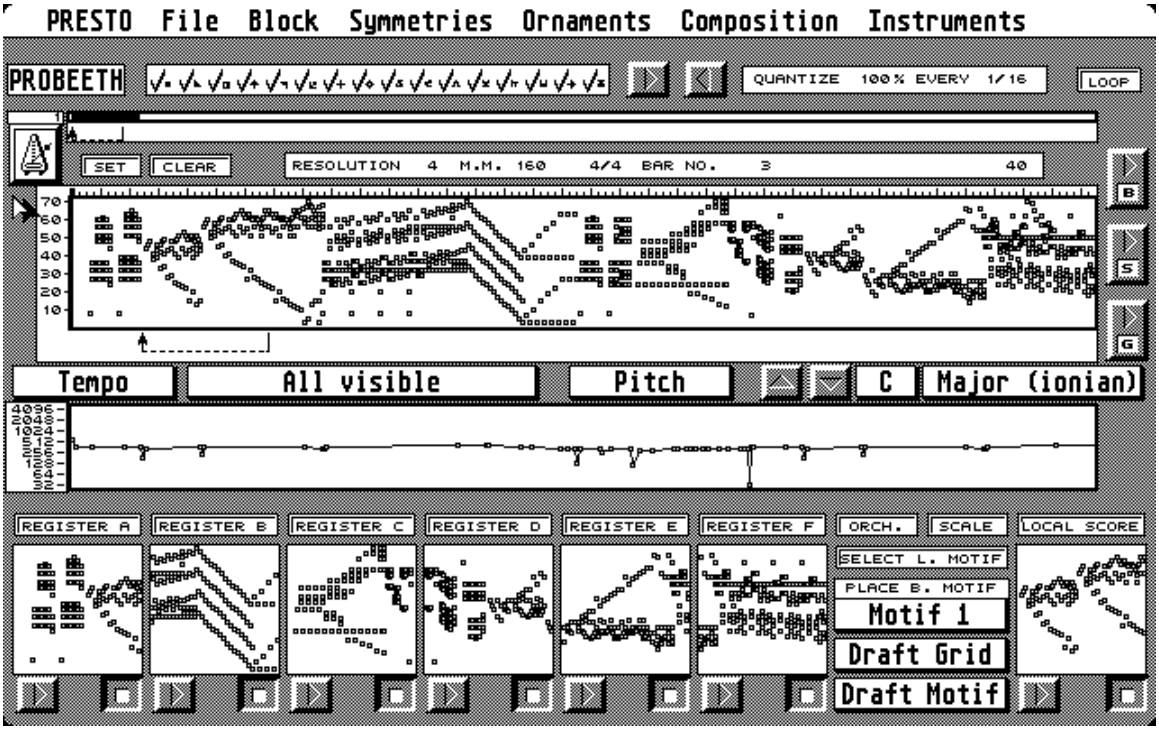


Figure 4.9: The Atari ST software *Presto*.

as alterations, where the notes of one composition where attracted by the notes of another. Furthermore, users could draw tempo curves and shape a piece's performance.

The latter is precisely what Mazzola and his collaborators extended in a later software, *Rubato*, for the NeXTSTEP platform (Figure 4.11). *Rubato* focused on analysis and performance and featured modular parts, so-called *rubettes*, for harmonic, metrical, and melodic analysis, with the results of which users could calculate weights for each note of a piece and let the system shape a performance of the piece based on these weights. *Rubato* was later adapted to the Mac OS X platform and some experiments were made to let the software interact with other software such as David Huron's *Humdrum* or IRCAM's *OpenMusic*. However, the software's dependency of

available in *Rubato Composer* and *BigBang*. Florian Thalmann. "Musical composition with Grid Diagrams of Transformations". Master's Thesis. University of Bern, 2007; Thalmann and Mazzola, "The BigBang Rubette: Gestural Music Composition with Rubato Composer".

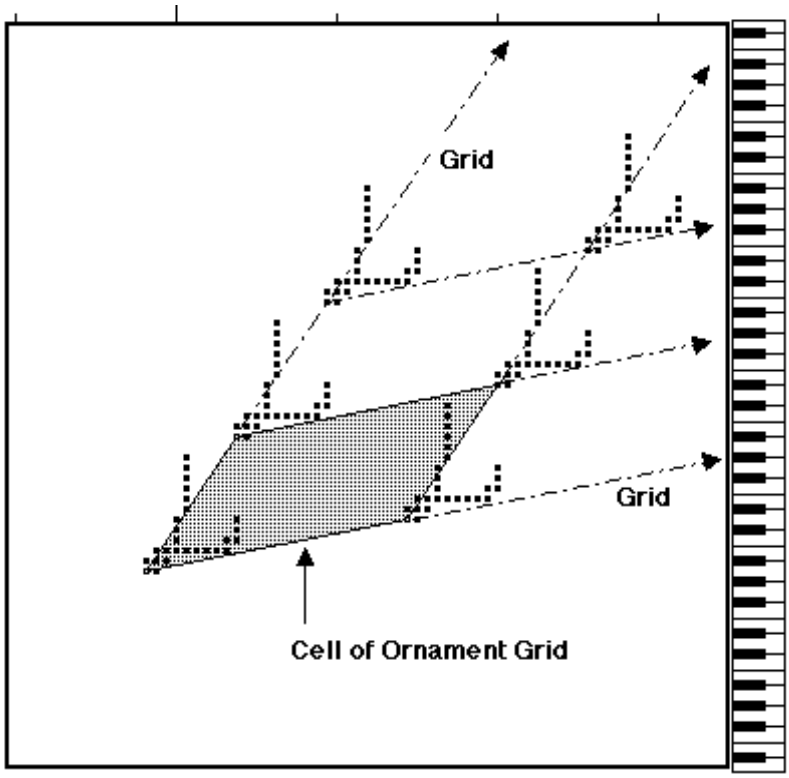


Figure 4.10: The *OrnaMagic* component of *Presto*.

Mac OS X and the lack of universal musical language led to the research group's decision to begin anew with the platform-independent language Java.

In a few early attempts, Stefan Müller, Stefan Göller, and Gérard Milmeister began with the implementation of a framework based on the denotator format (see Section 3.2.3). On top of this framework, Müller's *EspressoRubette* calculated performance fields (Figure 4.12) for given performances, in a way the opposite of what *Rubato* had done. Göller's system, the *PrimaVista* browser, visualized denotators in various ways in a three-dimensional space and let the user browse the space (see Section 6.1.1).

After these attempts, Mazzola's group felt the need for a much more modular system with small units of minimal functionality. This is how Gérard Milmeister's *Rubato Composer* came about. It was built on top of the partially developed frame-

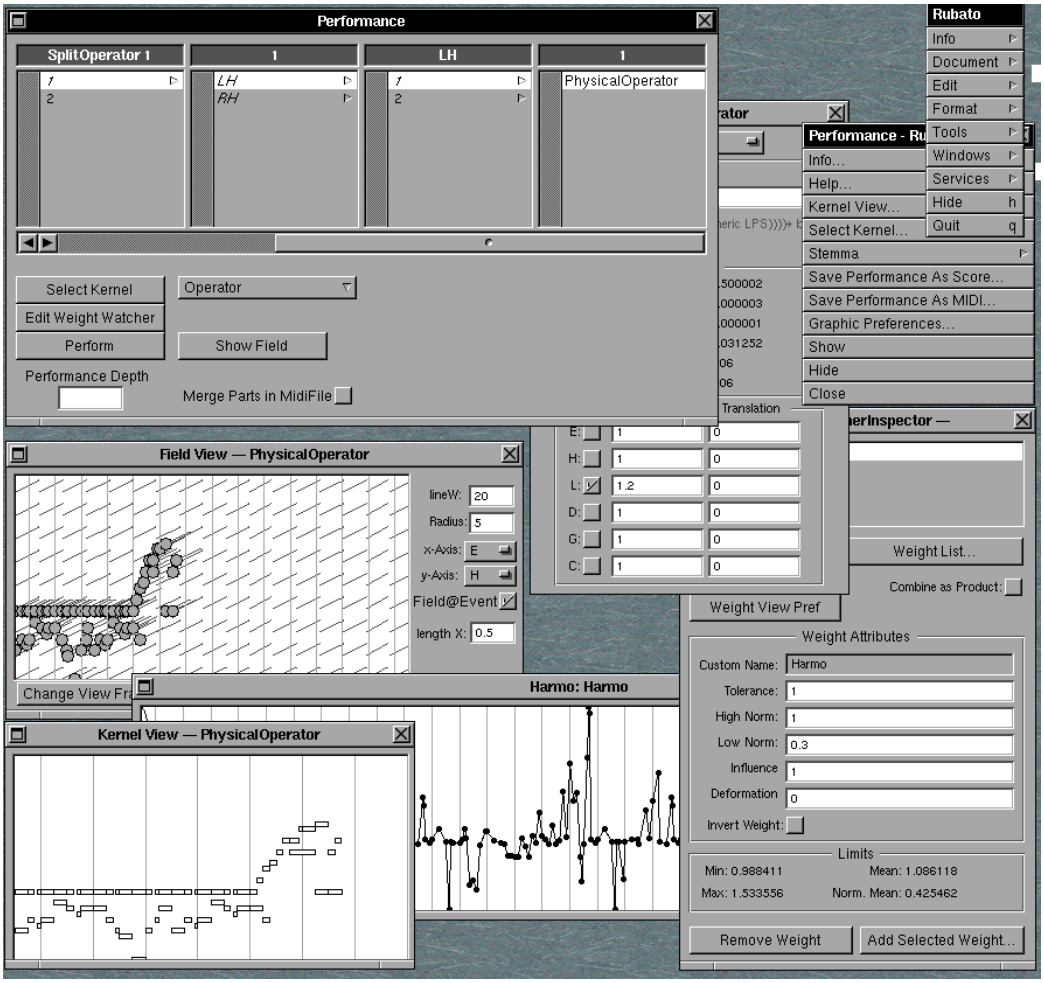


Figure 4.11: The *Rubato* software on NeXTSTEP.

work for forms and denotators and features a graphical user interface that allows users to compose, analyze, and perform music by creating networks of *rubettes*, in the fashion of *Max/MSP* or *Pd* (Section 4.3.2).

4.4.2 A Platform for Forms and Denotators

Figure 4.13 shows the overall architecture of *Rubato Composer*. The top level consists in the graphical user interface shown in Figure 4.14, whereas the lowest level shows the technology the software is based on, Java, along with subparts concerned with

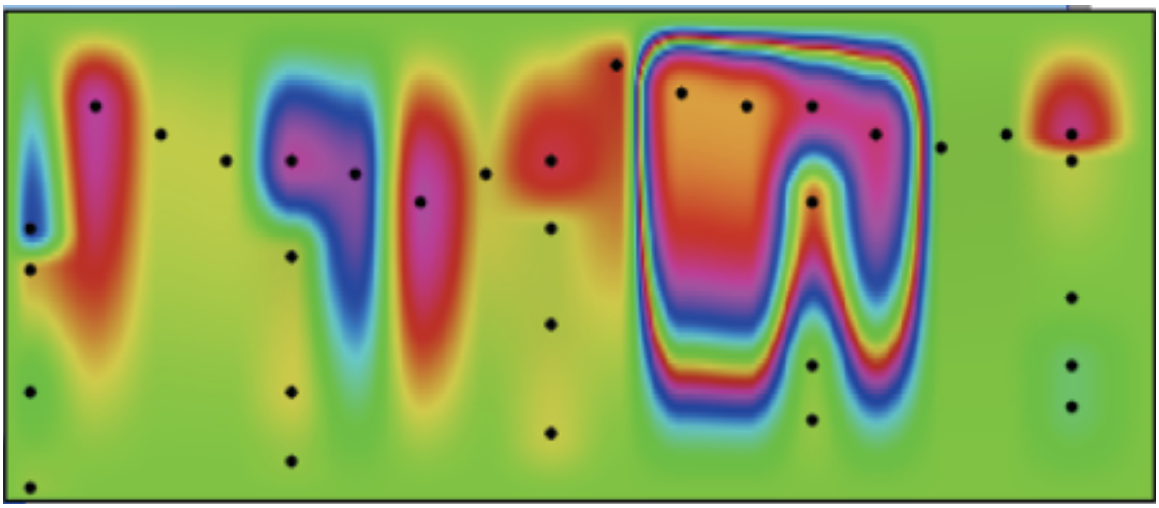


Figure 4.12: A performance field calculated by Müller's *EspressoRubette*.

the graphics (Swing), saving (XML), and communication (MIDI). In the meantime, there are interfaces to many other technologies or languages including CSound, JSyn, OpenGL, JPEG, LilyPond, or Leap Motion. Some of them will be described in Part II of this thesis.

The main concern in this section are the two middle levels in Figure 4.13. *Rubato Composer* includes a framework that implements the topos-theoretical constructs discussed in Section 3.2.3 of this thesis. For now, everything is based on the topos on the category of modules $\mathbf{Mod}^{@}$. Not only programmers but also users of the graphical interface can create any of the mathematical constructs imaginable in this topos. They can define modules of any dimension on the number rings $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, or $\mathbb{C}$, the modulo rings $\mathbb{Z}_n$, polynomial rings, and even string rings, to create words. In addition to this, they can define modules over product rings, such as $\mathbb{Q}^3 \times \mathbb{Z}_{12} \times \mathbb{C}$, as well as direct sum modules.[49] Then they can define Forms of any of the types **Simple**, **Limit**, **Colimit**, **Power**, or **List**, but **Limit** and **Colimit** only with trivial diagrams.

[49] Gérard Milmeister. *The Rubato Composer Music Software: Component-Based Implementation of a Functorial Concept Architecture*. Berlin/Heidelberg: Springer, 2009, p. 91.

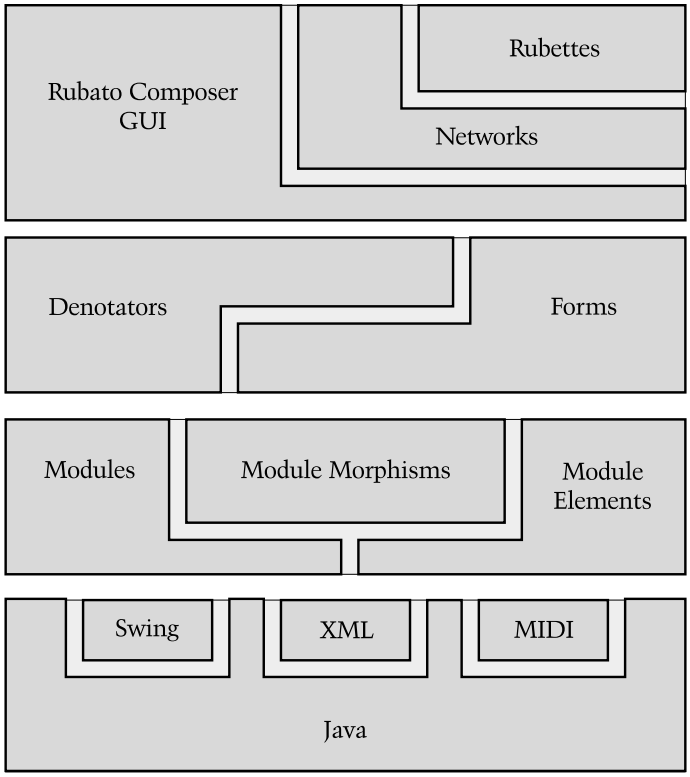


Figure 4.13: The architecture of *Rubato Composer*.

In order to create denotators of any of the forms created, users have to create a specific module elements for the modules in the **Simple** forms involved and build up the structure from there. Finally, these denotators can be mapped by morphisms, the domain of which are any of the involved modules. There is a great variety of available morphisms types, including embedding morphisms, reorder morphisms, split morphisms, and so on, which will not be relevant in the context of this thesis, as we will uniquely use affine morphisms.[50] These morphisms can again be combined into composed morphisms or product morphisms.

The framework provides helper classes for building forms and denotators. For forms, we can use the class `FormFactory`, as well as the appropriate classes for higher-

[50]For further reference, see Milmeister, *The Rubato Composer Music Software: Component-Based Implementation of a Functorial Concept Architecture*, p. 97ff.

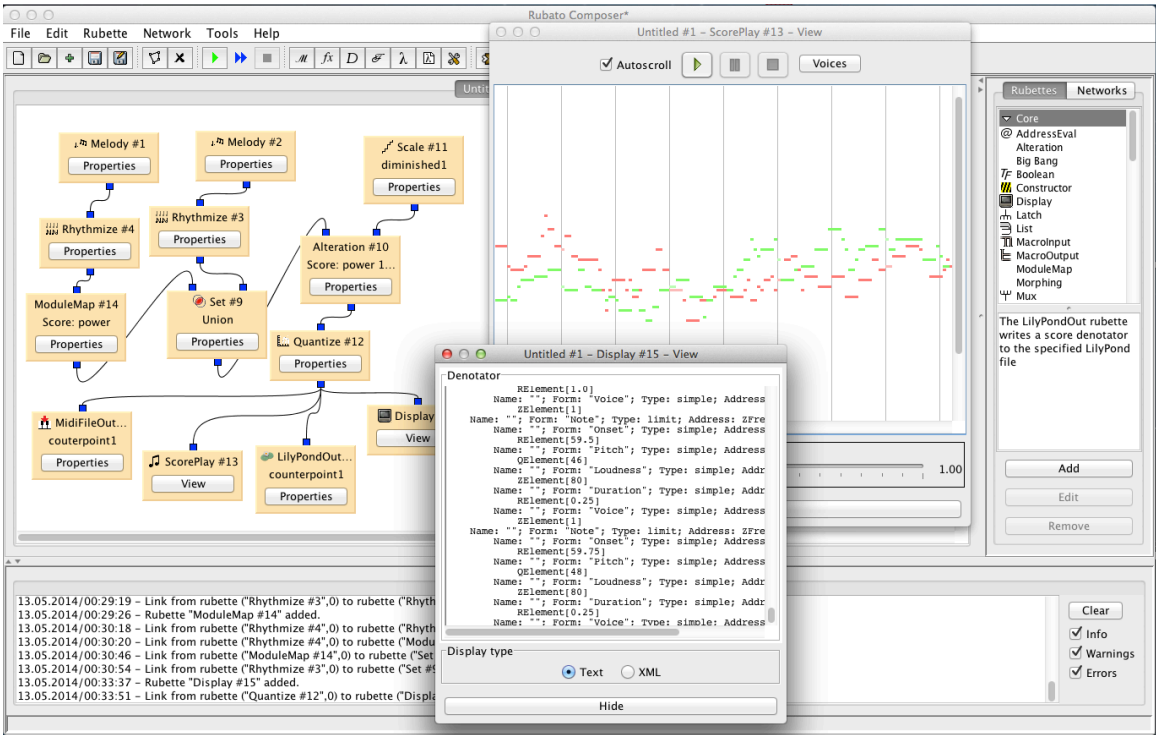


Figure 4.14: The graphical user interface of *Rubato Composer*.

dimensional modules. If, for instance, we wanted to build the *EulerScore* form defined in the example in Section 3.2.3 above, we could write the following code:

```
SimpleForm onset = FormFactory.makeQModuleForm("Onset");
Module eulerPitchSpace = QProperFreeModule.make(3);
SimpleForm eulerPitch = FormFactory
        .makeModuleForm("EulerPitch", eulerPitchSpace);
Module loudnessSpace = ZStringProperFreeModule.make(1);
SimpleForm loudness = FormFactory.makeModuleForm("Loudness", loudnessSpace);
SimpleForm duration = FormFactory.makeQModuleForm("Onset");
LimitForm eulerNote = FormFactory
        .makeLimitForm("EulerNote", onset, eulerPitch, loudness, duration);
LimitForm rest = FormFactory.makeLimitForm("Rest", onset, duration);
ColimitForm eulerNoteOrRest = FormFactory
        .makeColimitForm("EulerNoteOrRest", eulerNote, rest);
PowerForm eulerScore = FormFactory
        .makePowerForm("EulerScore", eulerNoteOrRest);
```

The corresponding denotators, also defined above, can then be created as follows.

We can use the class `DenoFactory` along with the appropriate module element classes. Note that all values are chosen to be exactly the same as in the example above. Also compare to Figure 3.1.

```
SimpleDenotator onset1 = DenoFactory.makeDenotator(onset, new Rational(0));
ModuleElement pitch1Element = ZProperFreeElement.make(new int[]{1, 0, -1});
SimpleDenotator pitch1 = DenoFactory
        .makeDenotator(eulerPitch, pitch1Element);
SimpleDenotator loudness1 = DenoFactory.makeDenotator(loudness, "sfz");
SimpleDenotator duration1 = DenoFactory
        .makeDenotator(duration, new Rational(1, 4));
Denotator note1 = DenoFactory
        .makeDenotator(eulerNote, onset1, pitch1, loudness1, duration1);
Denotator noteOne = DenoFactory.makeDenotator(eulerNoteOrRest, 0, note1);
```

```
SimpleDenotator onsetAtBeat2 = DenoFactory
        .makeDenotator(onset, new Rational(1, 4));
Denotator rest1 = DenoFactory.makeDenotator(rest, onsetAtBeat2, duration1);
Denotator shortRest = DenoFactory.makeDenotator(eulerNoteOrRest, 1, rest1);
```

```
SimpleDenotator onset2 = DenoFactory
        .makeDenotator(onset, new Rational(1, 2));
ModuleElement pitch2Element = ZProperFreeElement.make(new int[]{-1, 1, 1});
SimpleDenotator pitch2 = DenoFactory
        .makeDenotator(eulerPitch, pitch2Element);
SimpleDenotator loudness2 = DenoFactory.makeDenotator(loudness, "ppp");
SimpleDenotator duration2 = DenoFactory
        .makeDenotator(duration, new Rational(3, 2));
Denotator note2 = DenoFactory
        .makeDenotator(eulerNote, onset2, pitch2, loudness2, duration2);
Denotator noteTwo = DenoFactory.makeDenotator(eulerNoteOrRest, 0, note2);
```

```
Denotator twoNoteScore = DenoFactory
        .makeDenotator(eulerScore, noteOne, shortRest, noteTwo);
```

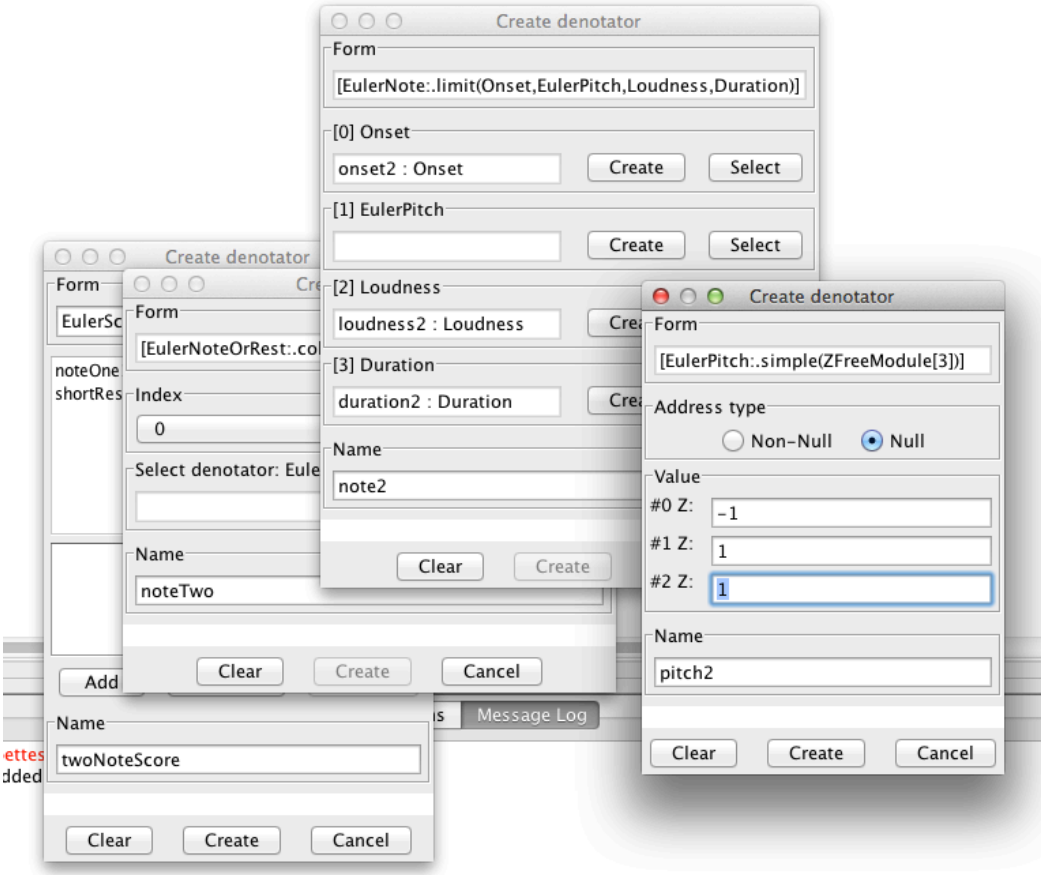


Figure 4.15: Creating the example *EulerScore* with *Rubato Composer*'s Denotator Builder

Forms and denotators can just as well be built when the software is running. For this, we can use all the dialog boxes offered by *Rubato Composer*'s graphical user interface. Figure 4.15 shows how this would have to be done for the same denotator as the one just defined in code.

Once they are built, all denotators, forms, and module elements can be accessed through paths. For instance, if we just had a reference to the above-defined *twoNoteScore*, we could obtain the *EulerPitch* denotator of its second note *noteTwo* by writing

```
Denotator pitchOfNoteTwo = twoNoteScore.get(new int[]{1,0,1});
```

The indices in the integer array argument stand for coordinate 1 of the *EulerScore* (noteOne is 0, shortRest is 2),[51] coordinate 0 in *EulerNoteOrRest* (1 would be for *Rests*), and coordinate 1 of *EulerNote* (which is *EulerPitch*). If we wanted to get the value of the dimension of thirds in the Euler space of the same note, we could write

```
int thirdValue = ((ZElement)pitchOfNoteTwo.getElement(new int[]{2}))
        .getValue();
```

This returns the third dimension of the element (index 2, first dimension or octaves have index 0, second or fifths have 1). There are similar methods that yield coordinators of forms.

Users can build all other mathematical constructs in **Mod**$^{@}$ in similar ways, all at runtime, which is a significant advantage to other applications. Whenever composers or improvisers have an idea of a new musical object, they can define it and start working with it right away. In the next section I will explain what working with denotators and forms looks like.

4.4.3 Rubettes and Networks

We have already seen how musical structures can be built in *Rubato Composer*. Now what can we do with them and how can we hear them? For this, we need to use the so-called *rubettes*, already mentioned in Section 4.4.1. In *Rubato Composer*, rubettes are small modular components that have a number of input and output connections through which they can be connected to other rubettes. Denotators travel through

[51] In a **Power** of a **Colimit**, elements are sorted by the colimit coordinate type, i.e. all instances of the first colimit coordinate, the all instances of the second, and so on.

the lines that are used to connect rubettes to each other. Each time the green *run* button in the top menu bar is pressed, the network is run in order, starting with the rubettes that receive no input, all the way to the rubettes that do not deliver an output. In Milmeister's opinion, this way, rubette networks visualize the work flow of the musician.[52] However, even though *Rubato Composer* networks can undoubtedly be considered processes, they do not follow the poietic process, since the users may define the bottom rubettes first. Much rather, they represent the processual logic behind the construction of the musical or analytical structure built.

The long-term goal of *Rubato Composer* is to provide a platform for analysis, composition, performance, improvisation, in short, any musical activity, and to do this with the constructs of mathematical music theory. It comes with a number of built-in rubettes with a variety of purposes. For instance, there are rubettes that transform denotators using arbitrary module morphisms (*ModuleMap*), perform set theoretical operations (*Set*), display denotators in text form (*Display*), transforms denotators of one form into ones of another (*Reform*), and so on.

In its early days, *Rubato Composer* was mainly used for processing denotators of a specific form, *Score*, simply a **Power** of *Notes*, which are in turn **Limit** of *Onset*, *Pitch*, *Loudness*, *Duration*, and *Voice*, all **Simple** of one-dimensional spaces. Several rubettes are designed to handle specifically *Scores*. *ScorePlay* plays them back, *Melody* creates melodies based on given characteristics, *Rhythmize* assigns generated rhythms to melodies, *Scale* generates scales, and *ScoreToCSound* exports them to CSound, and *MidiFileIn* and *MidiFileOut* load and save MIDI files to and from *Score* denotators. In general, however, rubettes should as much as possible be applicable to any sort of input. Some of the earlier *Score*-oriented operations available in the

[52]Milmeister, *The Rubato Composer Music Software: Component-Based Implementation of a Functorial Concept Architecture*, p. 76.

Presto software were generalized, for instance in form of the *Wallpaper* and *Alteration* rubette, which work for any arbitrary **Power** denotator.[53]

Figure 4.14 above contains a small network, where two melodies generated by *Melody* rubettes are rhythmized and the voice of one of them mapped via *ModuleMap* rubette. Then, they are combined via set union, altered towards an octatonic scale, quantized, and finally output as a LilyPond score and a MIDI file. The windows on the right of the network are the views of the *Display* rubette, showing denotators in text form, and the *ScorePlay* rubette, where the contrapuntal example can be played back so that it can be regenerated until satisfying.

Rubettes have to be implemented in Java and can easily be added as plugins to *Rubato Composer*. They can have a *properties* and a *view* window, the former of which can be used to set certain parameters needed to perform an action and the latter to visualize some desired aspects. A lot of functionality is provided through the classes `AbstractRubette` and `SimpleAbstractRubette`, the latter can be extended with just a few lines of code and comes with a customizable properties window. The main functionality of a rubette has to be implemented in its `run(RunInfo runInfo)` method, which is called by *Rubato Composer*'s main classes whenever a network is run, and where its properties can be accessed and its input denotators fetched, processed and assigned to the outputs.

For instance, we could create a rubette that generates the *EulerScore* defined above and sends it to all the connected rubettes. We would have to define the following constructor

```
public TwoNoteEulerScoreRubette() {
        this.setInCount(0);
        this.setOutCount(1);
```

[53]Thalmann, "Musical composition with Grid Diagrams of Transformations".

```
}
```

which means that the rubette only has one output and no input. The run method would then contain the code shown above followed by a simple line that assigns the score to the only output with index 0, thus

```
public void run(RunInfo runInfo) {
        SimpleForm onset = FormFactory.makeQModuleForm("Onset");
        ...
        Denotator twoNoteScore = DenoFactory
                .makeDenotator(eulerScore, noteOne, shortRest, noteTwo);
        this.setOutput(0, twoNoteScore);
}
```

4.4.4 Where are the Gestures?

Rubato Composer has an immense potential due to the versatility and completeness of its mathematical language, the infinite recombinations of its rubettes into networks, and the extensibility of its rubette collection. However, several of the distances discussed in the beginning of this thesis still apply and may limit its user group. Even though no programming skills are required for users that do not wish to write their own rubettes, the mathematical constructs and the ways in which they are created in *Rubato Composer* may discourage musicians, in ways similar to the application of mathematical music theory. The conceptual, generative, and sensual distances are maintained due to the abstractness of the interface. For instance, users have no chance to intuitively understand how a certain matrix produces a transformational result when defining affine morphisms in the *ModuleMap* rubette. The only distance *Rubato Composer* overcomes is the temporal distance. Upon any change of properties, the entire composition can be immediately recalculated. Even there, *Rubato Composer* has its limitations as it may take a long time and may be tedious to define

specific musical structures, so that the musicians lose their intuitive connection to the musical result and may stop being "inside" the music or stop experiencing flow.

Other systems built on the processual paradigm that allow for continuous data flow between their modular parts, such as *Max/MSP* or *Pd*, arguably overcome the *generative* distance, since a slight change in one of the parameters may lead to an immediately audible or visible result. However, for data structures as complex as the ones of *Rubato Composer*, there may need to be another solution. The idea behind this thesis is to provide a module for *Rubato Composer* that allows users to intuitively define, manipulate, visualize, and sonify any of the musical objects potentially definable within the mathematical framework. For this, after the above discussions, there is no better solution than the incorporation of gestures. The outcome is the *BigBang* rubette, which will be described in the remaining two parts of this thesis.

Part II

The BigBang Rubette and the Levels of Embodiment

Chapter 5

Introduction to BigBang

The *BigBang* rubette was developed as a solution to the above problems, with the goal to reduce the distances between the user, the mathematical framework, and the musical result. It has evolved over years, starting with a research project at the University of Minnesota in 2007. The early stages were described in several publications.[1] The subject of this thesis is a new, generalized version of *BigBang* that mainly evolved between 2012 and 2014 and that implements transformation-theoretical paradigms based on the ontological dimension of embodiment and the communication between these levels, as introduced in Part I of this thesis.

BigBang is a regular rubette with a number of view windows by means of which users can easily create denotators by drawing on the screen, visualize them from different perspectives, transform them in a gestural way, interact with a visualization of their compositional or improvisational process, and gesturalize the entire process in various ways. *BigBang* has one input and one output and the newest version accepts almost any type of denotator to be visualized and interacted with. Figure 5.1 shows

[1] For instance, Thalmann and Mazzola, "The BigBang Rubette: Gestural Music Composition with Rubato Composer"; Florian Thalmann and Guerino Mazzola. "Gestural Shaping and Transformation in a Universal Space of Structure and Sound". In: *Proceedings of the International Computer Music Conference*. New York City: International Computer Music Association, 2010.

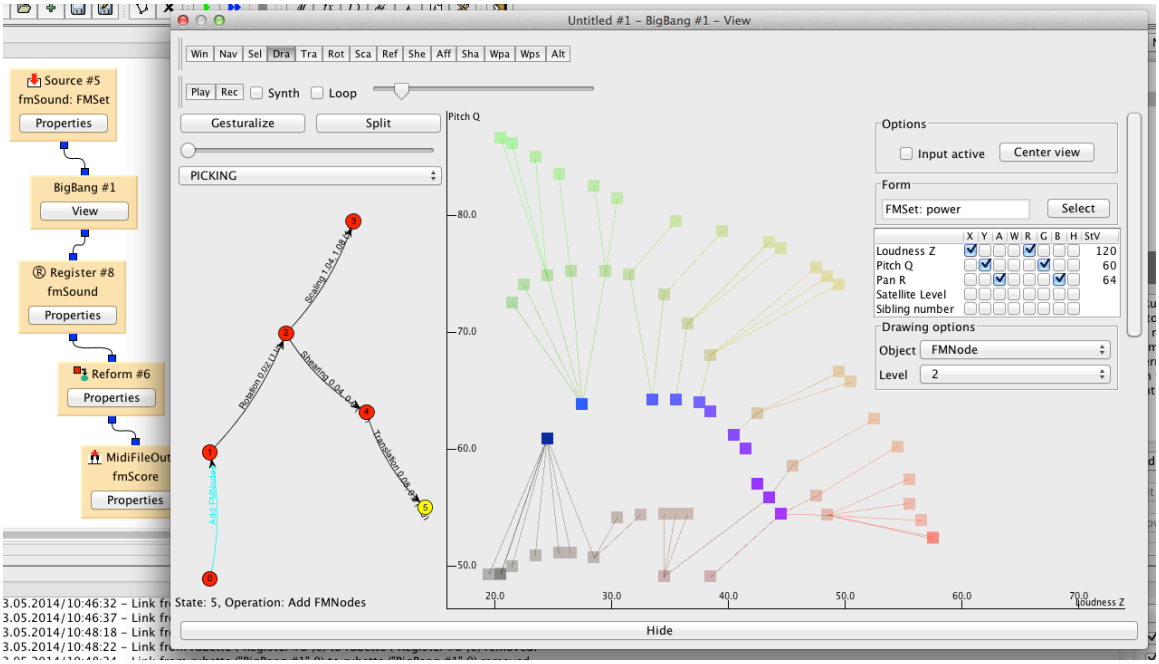


Figure 5.1: A network including the *BigBang* rubette and its view next to it.

the rubette, incorporated in a *Rubato Composer* network, and its view window.

All this is possible due to an implementation of the three levels of embodiment. The *facts view*, the large area on the right, visualizes the musical objects or facts, denotators in their coordinate space, and allows for different views of these objects, selectable using the grid of checkboxes on the right. The smaller area on the left, the *process view*, visualizes the process graph of the created music in the fashion of the graphs of transformational theory (Section 3.2). Each arrow corresponds to an operation or transformation, while each node corresponds to a state of the composition. Finally, the interaction with the objects visualized in the facts view is where the *gestures* happen. Any operation or transformation with *BigBang* is immediately and continuously sonified and visualized in a gestural way, according to the principles discussed in Section 4.3.3.

Otherwise, the rubette behaves as any other rubette: whenever the user presses on *Rubato Composer*'s run button, *BigBang* accepts a denotator, either adding it to

the one already present or replacing it, and sends its previous denotator to the next rubettes in the network. The rubette can be duplicated, which copies the graph in the process view along with any denotators created as part of the process. This way, users can include a *BigBang* with a defined process in other parts of the network, or other networks, and feed them with different inputs, while the process remains the same. Finally, as any rubette, *BigBang* can be saved along with the network, which again saves processes and corresponding facts.

The chapters in this part are structured according to the three levels of embodiment. In the first one, Chapter 6, I will deal with the facts view, explain how arbitrary musical objects are visualized, how they are sonified, and how they are represented within *BigBang*. Chapter 7 will be devoted to processes and explain what operations and transformations are available, how they can be applied, and how they are represented in *BigBang*'s process view. Finally, in Chapter 8, I will explain how the user's gestures are formalized, i.e. mapped onto transformations and operations, and how the resulting processes can be gesturalized again, so that users can see and hear their composition's evolution in a continuously animated way.

This part deals mainly with conceptual matters. A more detailed description of the architecture and implementation is given in Chapter 9 in Part III.

Chapter 6

Facts: BigBangObjects and their Visualization and Sonification

The facts, or objects, the *BigBang* rubette deals with, are denotators and the spaces in which these objects reside are forms, as discussed in the first part of this thesis. So far, we have only seen a small portion of the variety of forms that can be defined in *Rubato Composer*. However, any conceivable musical or non-musical object can potentially be expressed with forms and denotators, many of them in the category $\mathbf{Mod}^{@}$. One of the basic ideas of the advanced *BigBang* rubette described in this thesis was to generalize an earlier version that only worked with score-based denotators, similar to rubettes such as *ScorePlay* or *Melody*. The new *BigBang* was made compatible with as many forms as possible, even ones that the users may spontaneously decide to define at runtime. For this, not only *BigBang*'s *facts view* had to be generalized, but also the way denotators are represented internally, as of so-called `BigBangObjects`.[1] In this chapter, I describe how this was done.

[1]From now on, every object that literally exists as a Java object in *BigBang*'s code, will be written in verbatim font. However, they will mostly be described in a conceptual way. For more technical descriptions of the implementation, refer to Chapter 9.

6.1 Some Earlier Visualizations of Denotators

How can we define a visualization system that works for denotators of as many different forms as possible? First, it will be helpful to look at some earlier attempts at visualizing denotators. Several previous dissertations were based on an implementation of denotators and forms, as seen in Section 4.4.1. Stefan Göller's had visualization as its main focus and Gérard Milmeister's included a number of smaller visualization tools.

6.1.1 Göller's PrimaVista Browser

The goal of Göller's dissertation was to visualize denotators "in an active manner: visualization as navigation".[2] The result was the sophisticated *PrimaVista Browser*, implemented in Java3D, that featured a three-dimensional visualization in which users could browse denotators in first-person perspective. *PrimaVista* could be customized in many ways using a virtual device, the *Di*, shown in Figure 6.1.[3]

PrimaVista was capable of representing any type of zero-addressed $Mod^{@}$ denotator as a point or a set of points in $\mathbb{R}^3$ while preserving both order and distance of the original data structure as well as possible. **Limit** and **Colimit** denotators of any dimensionality and their nested subdenotators were folded in a two-step process, first into $\mathbb{R}^n$ then into $\mathbb{R}^3$. Thereby, for any denotator d the mapping $Fold : F(d) \to \mathbb{R}^3$ had to be injective. The first step of this process mapped the values of the **Simple** denotators found in the given denotator hierarchy, regardless of their domain, into $\mathbb{R}^n$ by injecting or projecting each of the individual values into $\mathbb{R}$. A matrix defined which denotator dimensions were mapped into which of the n dimensions of the real

[2]Stefan Goeller. "Object Oriented Rendering of Complex Abstract Data". Ph.D. Thesis. Universität Zürich, 2004, p. 55.

[3]Ibid., p. 107.

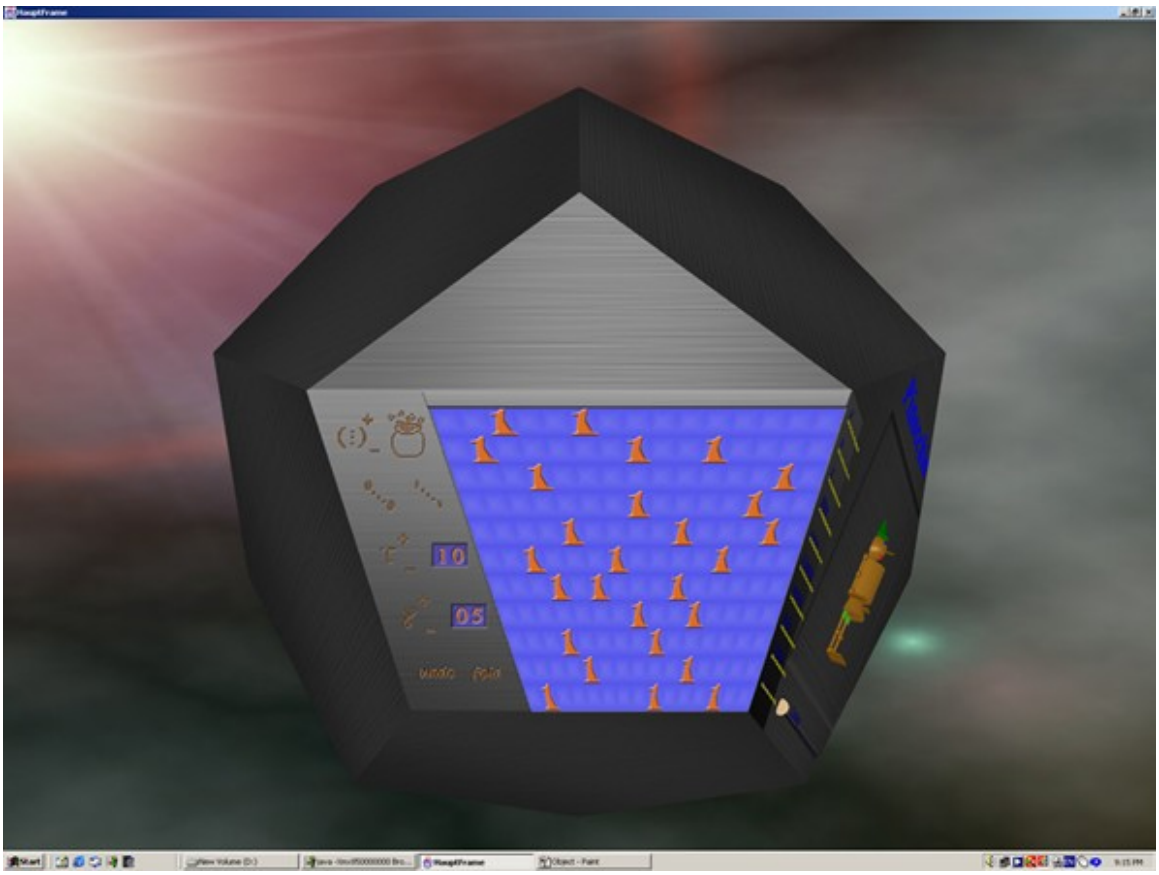


Figure 6.1: The *Di* of Göller's *PrimaVista* browser.

codomain space, allowing for both multiple mappings and merging mappings. A so-called *greeking* procedure made sure that only denotator values up to a certain level of hierarchical depth were taken into account, which enabled dealing with circular structures. The second step of the process consisted in folding the obtained $\mathbb{R}^n$ vectors into $\mathbb{R}^3$ by privileging specified dimensions and folding the remaining ones to the mantissa, the decimal digits after the comma.

Göller discussed adventurous ways of visualization replacing the points in $\mathbb{R}^3$ with complex three-dimensional objects the parts of which he called *satellites*, not to be confused with satellites as they are defined in this thesis,[4] each of them representing

[4]See Sections 6.2 and 9.2.3, where satellites are defined as elements of sub-powersets of a denotator. Also, in Göller's work, there are only two levels: the main satellite and its subsatellites.

additional characteristics of the represented denotators. Each of Göller's satellites is characterized by the following variable visual parameters: position (x, y, z), rotation vector (rx, ry, rz, α), scale (sx, sy, sz), color $(red, green, blue)$, texture, sound $(pitch, loudness, instrument, sysex)$.[5] The most complex object finally implemented is the Pinocchio satellite shown in Figure 6.2. Göller even suggests some satellites to be moving in time to represent parameters such as frequency. This feature was, however, finally not implemented. Another feature not implemented was a generalization of the musical score, where each satellite is associated with sounds that would be played when intersected with a plane, or more generally an algebraic variety, moving in time.[6] Finally, Göller discusses the concept of so-called *cockpits*, where an object's subsatellites become actuators in the form of levers, buttons, or knobs, through which users can change the underlying denotator.[7] Again, this was not implemented within the scope of his thesis. In addition to this, Göller envisioned ways of transforming and manipulating objects that are similar to the ones of the *BigBang* rubette.[8]

There are several issues with Göller's approach, some of which explain the difficulties that arose when trying to implement the ideas. First, the folded spaces pose problems of ambiguity in visualization and especially transformation. If one dimension of $\mathbb{R}^3$ represents several denotator dimensions at the same time and the user starts transforming the denotator, it is not intuitively deducible from the visible movement how the denotator values are affected. Representation is often ambiguous, where differences in dimensions folded to the mantissa become only subtly visible and often visually indistinguishable from a simple projection. Second, several simplifications of the denotator concept were made to enable representation within this model. Göller

[5] Goeller, "Object Oriented Rendering of Complex Abstract Data", p. 77.
[6] Ibid., p. 84-5.
[7] Ibid., p. 95.
[8] Ibid., p. 123f.

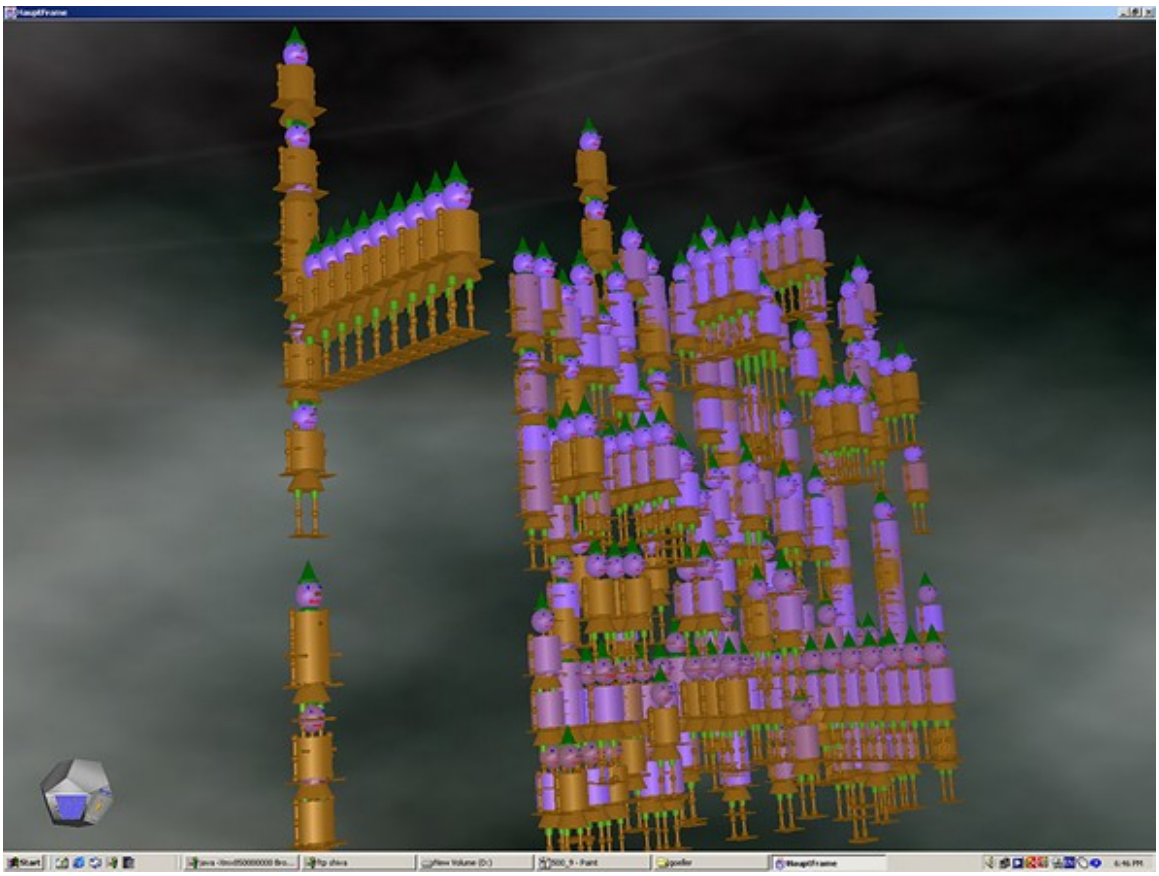


Figure 6.2: A denotator visualized in *PrimaVista* using Pinocchios (satellites) of varying size and differently positioned extremities (subsatellites).

does, for instance, not consider higher-dimensional **Simple** forms, such as ones using modules based on $\mathbb{R}^2$ or $\mathbb{C}$. Third, he mainly visualizes denotators on the topmost level, thereby assuming that it consists of a **Power**.[9] The BigBang rubette offers solutions to several of these problems, as discussed later.

6.1.2 Milmeister's ScorePlay and Select2D Rubettes

Even though the focus of Milmeister's work lay in building the basic mathematical framework as well as the interface of *Rubato Composer*, some of his rubettes offer visualizations of denotators of both general and specific nature. The *ScorePlay* rubette

[9] Goeller, "Object Oriented Rendering of Complex Abstract Data", p. 63.

limits itself to *Score* denotators, as mentioned above, and represents them in piano roll notation, as can be seen above, in Figure 4.14. It simply visualizes a *Score* and enables users to play it back at a variable tempo and using different built-in MIDI instruments. It does not allow for any interaction with the represented notes.

The *Select2D* rubette represents any incoming **Power** or **List** denotator as points projected to a customizable two-dimensional coordinate system, the axes of which can be freely associated with any **Simple** denotator somewhere in the denotator hierarchy. Users can then select any number of these points by defining polygons around them (Figure 6.3). The rubette then outputs the subdenotators associated with these points as one runs the network.

Milmeister's rubettes provide several improvements over Göller's software while being more limited in other ways. *ScorePlay* only accepts denotators of one form and visualizes them rigidly. However, its visualization is minimal and based on a standard immediately understandable by the user, which Göller's might not always be. *Select2D*, in addition to **Power** denotators, also accepts **List** denotators, which were only introduced in Milmeister's work.[10] Furthermore, it is able to represent more types of **Simple** denotators than Göller's, more precisely ones containing free modules over any number ring except for $\mathbb{C}$. Nevertheless, higher-dimensional **Simple** coordinates and product rings can again not be represented. Furthermore, the rubette's visualization capabilities do not exceed the representation of points projected to a two-dimensional coordinate system.

[10]Milmeister, *The Rubato Composer Music Software: Component-Based Implementation of a Functorial Concept Architecture*, p. 105.

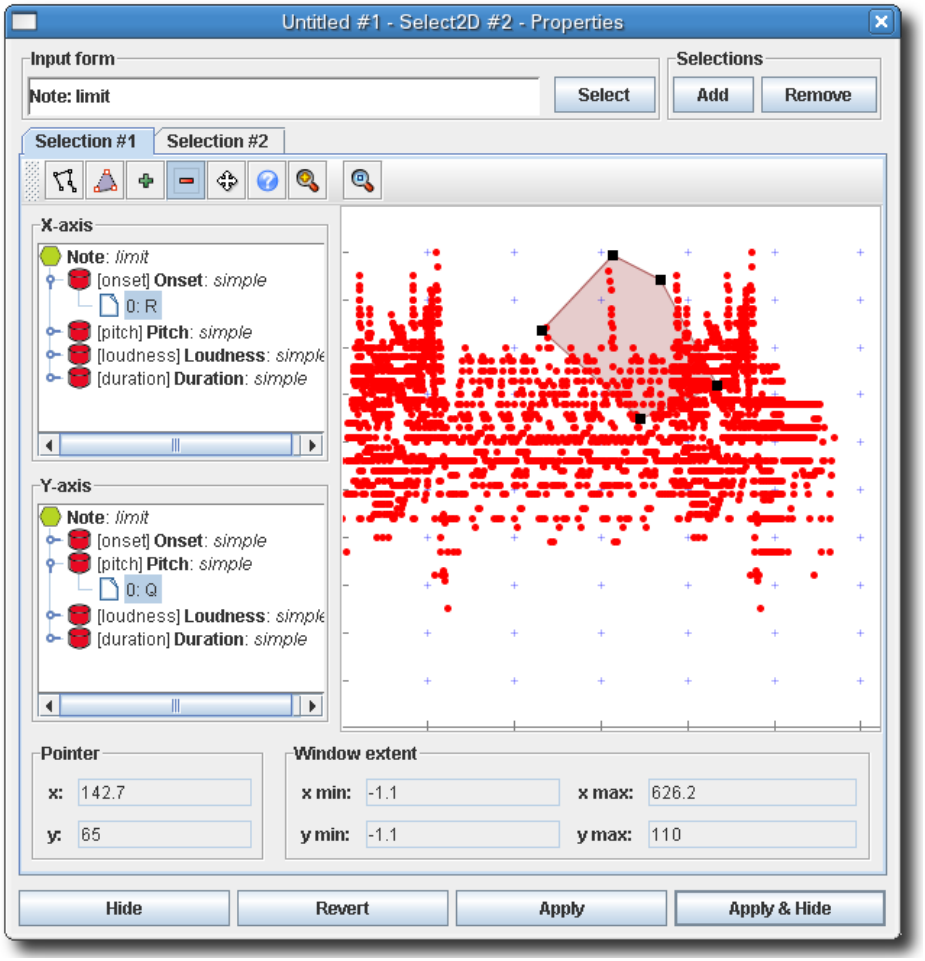


Figure 6.3: The *Select2D* rubette showing a *Score* denotator on the *Onset* × *Pitch* plane.

6.2 An Early Score-based Version of BigBang

Initially, the *BigBang* rubette was designed for a small set of score-related denotators. The first version allowed users to handle *Scores* and *MacroScores* and was developed before in the context of an independent research project at the University of Minnesota.[11] *MacroScore* is a conceptual extension of the form *Score* which I casually defined in Section 4.4.3. It brings hierarchical relationships to *Notes* by imitating the

[11] Thalmann and Mazzola, "The BigBang Rubette: Gestural Music Composition with Rubato Composer".

set-theoretical concept of subsets.[12] The form is defined in a circular way, as follows:

$$MacroScore : .\mathbf{Power}(Node),$$

$$Node : .\mathbf{Limit}(Note, MacroScore),$$

$$Note : .\mathbf{Limit}(Onset, Pitch, Loudness, Duration, Voice)$$

Each *Node* associates thus a *Note* with a set of again *Nodes*, each of which again contain a *Note* and a set, and so on. In short, with this construction, each *Note* of a *MacroScore* has a set of so-called *satellites* on a lower hierarchical level. We could go on infinitely, but in order to stop at some point, we give some of the *Nodes* empty sets, thus no satellites. The idea behind this form is that in music, we not only often group objects together and wish to treat them as a unity, but also establish hierarchies between them. A trill, for instance, consists of a main note, enhanced by

[12] This complies Graeser's notion of counterpoint as "a set of sets of sets of notes", cited in Note 1 in Section 3.1.

some ornamental subnotes.[13] A simplified trill denotator could be defined as follows:

$$
\begin{aligned}
&shakeWithTurn : @MacroScore(mainNode),\\
&mainNode : @Node(mainNote, ornamentalNotes),\\
&mainNote : @Note(\ldots),\\
&ornamentalNotes : @MacroScore(\\
&\quad upNode, midNode, upNode, midNode, lowNode, midNode),\\
&upNode : @Node(upNote, emptySet),\\
&upNote : @Note(\ldots),\\
&emptySet : @MacroScore(),\\
&\ldots
\end{aligned}
$$

What is crucial to the notion of satellites, is that their values are defined relatively to the ones of their anchor. So if for instance the *mainNote* defined above has *Pitch* 60 and its satellite *upNote Pitch* 61, the latter in fact obtains a *Pitch* of 1. If another had *Pitch* 58 it would be defined as -2. This way, if we transform the anchor, all its satellites keep their relative positions to it.

Later on, another form was added to *BigBang*'s vocabulary, *SoundScore*, which combines frequency modulation synthesis with the *MacroScore* concept. Each note, in addition to having satellites, can have modulators which modulate its frequency and change its timbre.[14] Again, modulators have a relative position to their carrier

[13] In a similar way, Schenkerian analysis describes background harmonic progressions enhanced by ornamental foreground progressions, which could be represented with *MacroScores* as well. However, we may find forms that are better suited, as will be discussed below.

[14] Thalmann and Mazzola, "Gestural Shaping and Transformation in a Universal Space of Structure and Sound".

and would be transformed with it. The form is defined as follows:

$SoundScore$: .**Power**($SoundNode$),

$SoundNode$: .**Limit**($SoundNote$, $SoundScore$),

$SoundNote$: .**Limit**($Onset$, $Pitch$, $Loudness$, $Duration$, $Voice$, $Modulators$),

$Modulators$: .**Power**($SoundNote$)

Denotators of these forms are all based on the same five-dimensional space spanned by the **Simple** forms *Onset*, *Pitch*, *Loudness*, *Duration*, and *Voice* and can thus be visualized the same way. The early *BigBang* rubette did this using a generalized piano roll representation, as I will explain later on.[15] In sum, all of the objects the early *BigBang* rubette dealt with were essentially notes.

6.2.1 The Early BigBang Rubette's View Configurations

The visualization principle of the BigBang rubette[16] combines elements of both Göller's and Milmeister's models, but focuses on a minimalist appearance aiming towards simplicity and clarity. It generalizes the piano roll notation also used in the ScorePlay rubette (see Section 6.1.2). Notes are represented by rectangles on a two-dimensional plane, just as in a piano roll. However, already in early versions of BigBang, the visual elements of the piano roll were separated from their original function so that they could be arbitrarily assigned to the symbolic dimensions of the represented score denotator. This is reminiscent of the ways Göller's subsatellites could be assigned to any folded denotator dimensions (Section 6.1.1) or of the spacial representation of Milmeister's *Select2D* rubette (Section 6.1.2). A similar method of visualizing was

[15] Piano roll is a standard in music software, as mentioned in Section 4.3.1.

[16] Thalmann and Mazzola, "The BigBang Rubette: Gestural Music Composition with Rubato Composer", p. 4-5.

also available in *Presto*'s local views (Section 4.4.1).

In order to do this I defined a set of six visual parameters

$$N = \{X\text{-}Position, Y\text{-}Position, Width, Height, Opacity, Color\}$$

corresponding to the visual properties of piano roll rectangles along with a set of six note parameters

$$M' = \{Onset, Pitch, Loudness, Duration, Voice, SatelliteLevel\},$$

which corresponds to the **Simple** denotator in *Scores* with the exception of $SatelliteLevel$, which was used to capture the hierarchical level of satellite notes in *MacroScores* and *SoundScores*. I then defined a *view configuration* to be a functional graph $V \subset N \times M'$. This ensures that each screen parameter $n \in N$ is associated with at most one musical parameter $V(n)$ that defines its value, as well as that V does not need to include all $n \in N$. View parameters not covered by V obtain a default value that can be defined by the user. The traditional piano roll notation could be produced by selecting the following pairing (shown in Figure 6.4):

$$V_1 = \{(X\text{-}Position, Onset), (Y\text{-}Position, Pitch), (Width, Duration)\}$$

An enhanced version of the piano roll that often appears in software products also uses opacity and color:

$$V_2 = \{(X\text{-}Position, Onset), (Y\text{-}Position, Pitch), (Opacity, Loudness), \\ (Width, Duration), (Color, Voice)\}$$

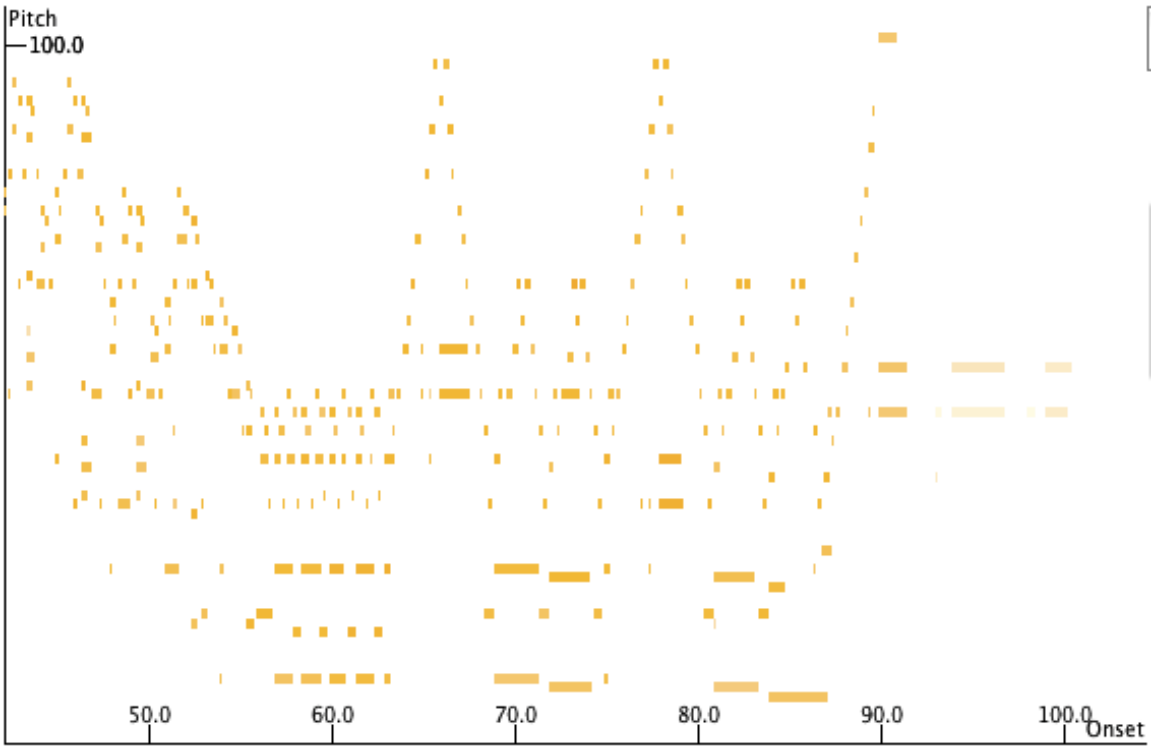


Figure 6.4: The early *BigBang* rubette showing a *Score* in piano roll notation.

The possibility of arbitrary pairings, however, also enables more adventurous but possibly also interesting view configurations, such as the following (Figure 6.5):

$$V_3 = \{(X\text{-}Position, Onset), (Y\text{-}Position, Loudness), (Width, Pitch),$$
$$(Color, Onset), (Height, Loudness)\}$$

Experimenting with such view configurations may be especially valuable for analysis and may lead to a different understanding of given musical data sets.

Every view parameter can be customized at runtime. Depending on the represented note parameter, it can be useful to ensure that a screen parameter's value does not exceed a specific value range. For example it may look more clear when the rectangle's heights are limited in a way that their areas do not intersect, just as

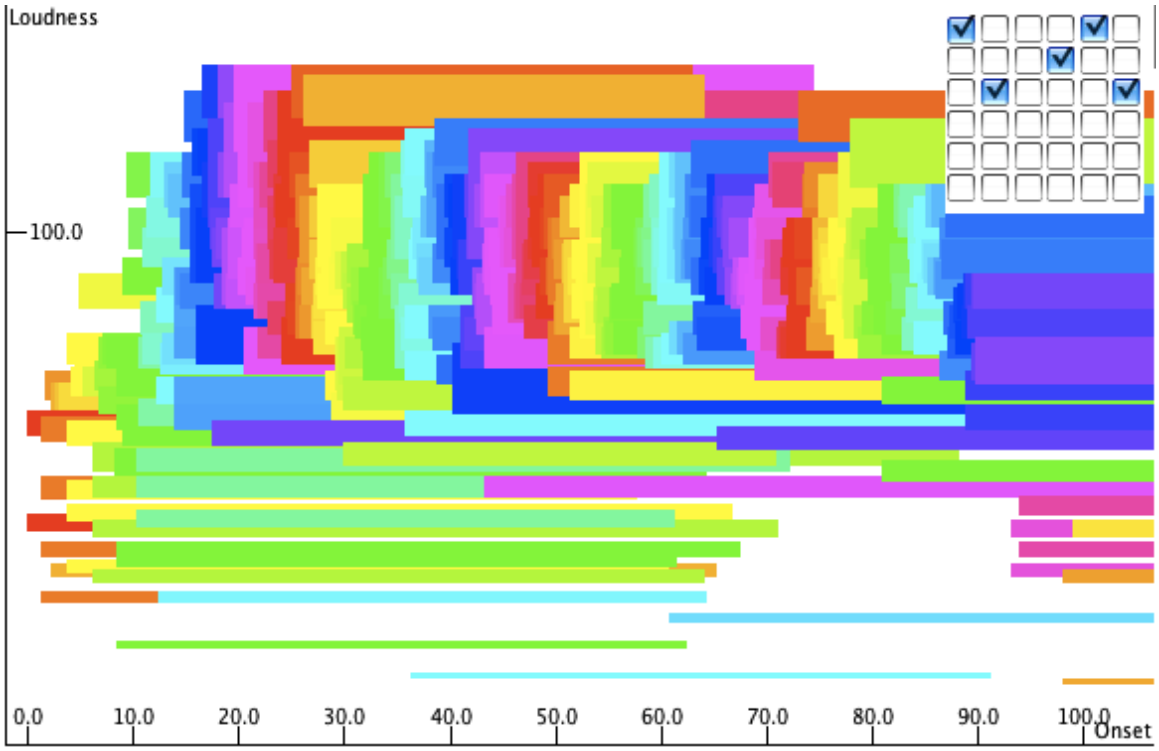


Figure 6.5: The early *BigBang* rubette visualizing a *Score* in a more experimental way.

with piano roll notation. Thus, for each $n \in N$, we optionally define min_n, max_n, the minimal and maximal screen values. We then have two options to define the way note parameters are mapped to the screen parameters.

1. If we choose the conversion to be *relative*, the minimal and maximal values of the given note parameter $min_m, max_m, m \in M'$ are determined for the actual score, and then mapped proportionally so that the note with min_m is represented by min_n and the note with max_m by max_n. For this, we use the formula

$$v_n = \frac{v_m - min_m}{max_m - min_m}(max_n - min_n) + min_n,$$

where v_n is the screen value for the note value v_m.

2. On the other hand, *absolute* mapping means that every value with $v_m < min_n$ or $v_m > max_n$ is mapped to a new value, while all other values stay the same, i.e. $v_n = v_m$. For absolute mapping, we have two choices. In *limited* mapping, the values that surpass the limits are given the min_n and max_n values, respectively. The following formula is used:

$$v_n = \begin{cases} min_n, \text{ if } v_m < min_n \\ max_n, \text{ if } v_m > max_n \\ v_m \text{ otherwise.} \end{cases}$$

For *cyclical* mapping, we use the formula

$$v_n = \begin{cases} (v_m \mod (max_n - min_n)) + min_n, \\ \quad \text{if } v_m < min_n \text{ or } v_m > max_n \\ v_m \text{ otherwise.} \end{cases}$$

This mapping type can be useful for the color screen parameter for example, where it is reasonable to cycle through the color circle repeatedly to visualize a specific note parameter interval, such as an octave in pitch, or a temporal unit, as shown in Figure 6.5, where color visualizes a time interval of length 24, i.e. six 4/4 bars.

With absolute mapping it is possible to leave either or both of the **Limit**s as undefined. Accordingly, we assume $min_n = -\infty$ or $max_n = \infty$. Of course, if none of the limits are defined, the visible screen parameters correspond exactly with the original note parameters.

At runtime, the view window's current pairings could be selected using a *matrix*

of checkboxes with a column for each screen parameter and a row for each note parameter, see Figure 6.5.

Satellite relations can be displayed in two ways. First, the note parameter *SatelliteLevel*, mentioned above, can be assigned to any arbitrary visual parameter. This way, anchor notes are associated with integer value 0, first-level satellites with 1, and so on. On the other hand, satellite relations may also be displayed as lines between the centers of two note objects so that every note has lines leading to each of its direct satellites, as shown in Figure 6.6.[17] As mentioned above, since all anchors and satellites in *MacroScore* and *SoundScore* denotators are notes, they can be represented in the same space.

6.2.2 Navigating Denotators

Users can navigate this two-dimensional space not only by changing their view of the space by choosing different note parameters for the x and y view parameters, but also by changing their view point by scrolling the surface and zooming in and out without limitations. This is similar to Göller's *PrimaVista*, but using two instead of three dimensions. However, users can also open several of these views simultaneously and choose different perspectives on the composition. This is especially valuable when performing transformations in one view while observing how the composition is affected from the other perspective.

6.2.3 Sonifying Score-based Denotators

In early *BigBang*, denotators could not only be visualized but also sonified. Even though this may be done using another, specialized rubette such as *ScorePlay*, we

[17] This is a notion of satellites significantly different from Göller's (Section 6.1.1). While Goeller uses the term to denote movable parts of objects and represents them as denotators, but here we use it to speak of circular denotator structures (also see Note 4).

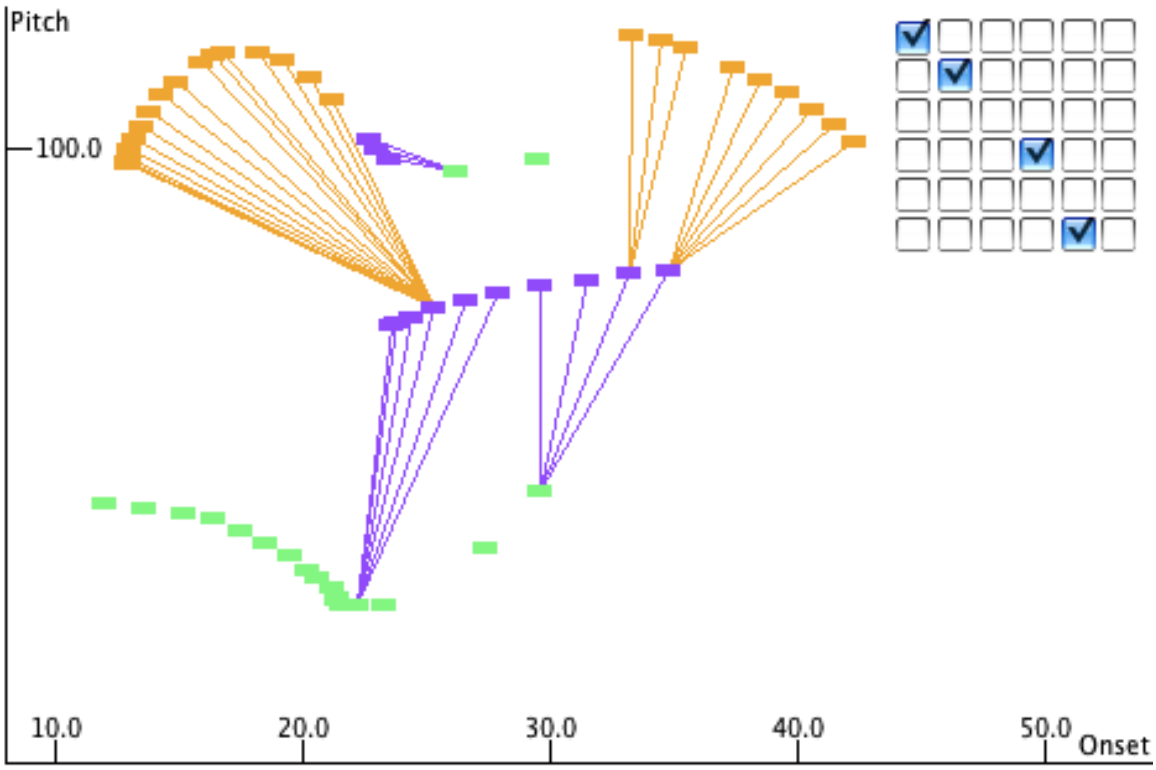


Figure 6.6: The early *BigBang* rubette showing a *MacroScore* with two levels of satellites.

decided to include this functionality within *BigBang*. The main reason for this was the gestural interaction concept, where especially immediate auditive feedback is key, as we saw in Section 4.3.3. Users have to be able to judge musical structures by ear while they are creating them, and the use of an external rubette would have slowed down the process. A second reason was that many of the possible musical structures in early *BigBang* were micro-tonal for which MIDI feedback, as implemented in *ScorePlay*, is unsuited since it is strictly chromatic.[18] The extension of *BigBang* for *SoundScores* was another reason, for now timbre was part of the musical objects and had to be judged while it was defined.

Since all the structures dealt with in early *BigBang* were *Score*-based denotators,

[18]The use of pitch bend is an option for monophonic material, but limited as soon as several notes have to be bent in different ways.

sonification was rather straightforward. All the objects that had to be played were *Notes* that existed in the same space. They were simply played back in time, giving the user control over tempo. The microtonal and frequency modulation structures of *SoundScores* made it necessary for a synthesizer to be used. For each note, a synthesizer object, a so-called `JSynNote`[19] was created by converting symbolic time, pitch, and loudness into the physical parameters time, frequency, and amplitude.

Outside *BigBang*, *MacroScores* usually have to be converted into *Scores* in order to be played back, a process called *flatten* (see next chapter). In early *BigBang*, this happened in the background, since it would have significantly slowed down the composition process. Satellites were simply converted into additional `JSynNotes` accordingly. Modulators in *SoundScores*, however, became modulators of `JSynNotes`. There were two options of how to play back modulators: either their temporal parameters were ignored and they simply played whenever their carrier was playing, or they only modulated their carrier according to their own onset and duration. In the latter case, users had to make sure the anchor notes were playing at the same time as their modulators, but they also had the chance to create temporally varying configurations of modulators for a single note.

6.3 BigBangObjects and Visualization of Arbitrary $Mod^{@}$ Denotators

Despite its customizability, the view concept of the early *BigBang* rubette was first designed to represent *Score*, *MacroScore*, and *SoundScore* denotators, which are all based on the same musical space: (**Power** of...) **Power** of **Limit**. There, the view concept has proven its viability, compared to other concepts such as the ones

[19] JSyn is the name of the synthesizer framework we decided to use, as will be explained later on.

discussed above. Now how can this concept be generalized so that *BigBang* can accept any denotators?

In this section we describe how we can do this for a major part of the spaces available in Milmeister's version of Rubato Composer, which are all based on elements of the topos $Mod^{@}$ over the category of modules, as described in Section 4.4.2.[20] The number of denotator types capable of being represented by the new *BigBang* rubette is significantly higher than the two comparable modules *PrimaVista* and *Select2D*. Nevertheless, for the time being we restrict ourselves to 0-addressed denotators (see Section 3.2.3) and focus on number-based modules. We exclude both modules based on polynomial rings and ones based on string rings, since their visualization may differ markedly and will be left to future projects.

6.3.1 A Look at Potential Visual Characteristics of Form Types

As a starting point we need to reflect on the role of the five form types **Simple**, **Limit**, **Colimit**, **Power**, and **List** and the way they can best be visualized. Each of these types implies another visual quality that may be combined with the others. How were these qualities in early *BigBang*? *Scores* were shown as clusters of rectangles (**Power**) within a coordinate system (**Limit**) of five axes (**Simple**), which could in turn be variably shown as any of six visual dimensions (x-position, y-position, width, height, color, opacity). Three of the five form types are involved here. The **Simple**s in a *Note* are based on free modules on a number ring and can thus easily be represented by one number axis or one of the other visual properties. However, *Rubato Composer*

[20] This procedure is also described in Florian Thalmann and Guerino Mazzola. "Visualization and Transformation in a General Musical and Music-Theoretical Spaces". In: *Proceedings of the Music Encoding Conference 2013*. Mainz: MEI, 2013, p. 3, as well as briefly in Florian Thalmann and Guerino Mazzola. "Using the Creative Process for Sound Design based on Generic Sound Forms". In: *MUME 2013 proceedings*. Boston: AAAI Press, 2013.

allows for many more types of **Simple**s, each of which must be consider here:

Simple Denotators

Simple denotators are crucial to a system of visualization, since they are the only denotators that stand for a specific numerical value in a space. Basically, every form that will be used in a practical way should to contain **Simple**s. This despite the fact that it is possible to conceive more pathological forms, such as for instance the circular form that describes sets as sets of sets:

$$Set : .\mathbf{Power}(Set).$$

Such forms will be of little use in our context, since anything to be represented and especially transformed needs to contain specific numerical values. We can thus declare a first rule here:

Rule 1 *In our system denotators will only be represented if they contain at least one* **Simple** *denotator somewhere in their structure.*

With the system, **Simple** denotators over the following modules can be represented:

Free Modules over Number Rings The most straightforward type of modules are the free modules based on number rings such as $\mathbb{Z}, \mathbb{Q}, \mathbb{R}$, or $\mathbb{C}$. Elements of the former three are typically represented along an axis, whereas ones of the latter on a two-dimensional Cartesian system. For modules $\mathbb{Z}^n, \mathbb{Q}^n, \mathbb{R}^n$, and $\mathbb{C}^n$ an n-dimensional or $2 * n$-dimensional system of real axes will be appropriate.

Furthermore, as shown in Section 6.2.1, as long as all values of a specific denotator are known and finite, dimensions of free modules over number rings can equally be represented by other visual parameters, such as an objects width, height, color, etc.

Elements of the free module over $\mathbb{C}$, for instance, could convincingly be represented as width and height of objects.

Quotient Modules For free modules over quotient modules such as $\mathbb{Z}_m = \mathbb{Z}/m\mathbb{Z}$, $\mathbb{Q}_m = \mathbb{Q}/m\mathbb{Z}$, $\mathbb{R}_m = \mathbb{R}/m\mathbb{Z}$, and $\mathbb{C}_m = \mathbb{C}/m\mathbb{Z}$ we choose a manner of representation that corresponds to the one introduced in the previous section, where values are simply projected on a one- or two-dimensional coordinate system. However, instead of being potentially infinite, the axes maximally show the numbers of the interval $[0, m[$, which makes zooming out beyond this point impossible. This works in an analogous way for other view parameters that do not allow cyclical representation, such as width, height, and opacity. Of the defined visual parameters, only color allows for cyclical representation, in analogy to the color circle. Again, for $\mathbb{Z}_m^n, \mathbb{Q}_m^n, \mathbb{R}_m^n$, and $\mathbb{C}_m^n$, the system can be extended to be n-dimensional or $2 * n$-dimensional.

Modules over Product Rings and Direct Sums of Modules over Quotient Modules Representation is straightforward for direct sums of any of the quotient modules discussed so far. Each of the factors is independently associated with one of the view parameters. For example, for $\mathbb{Z} \times \mathbb{R}_7$ we might choose to represent the $\mathbb{Z}$ part with the x-axis and the $\mathbb{R}_7$ part with color.

Remark: More general modules which are not derived from direct sums of such quotient modules are not yet dealt with.

Limit Denotators

The fact that **Limit**s are products or conjunctions makes them always representable in the conjoined space, i.e. the cartesian product of the spaces of their factors. The most simple case is a **Limit** of **Simple** denotators, just as with our common *Score* denotators. *Notes* can be represented in $Onset \times Pitch \times Loudness \times Duration \times$

Voice. This is even possible if the same subspaces appear in several times. For instance, if we define a form

$$Dyad : .Limit(Pitch, Pitch),$$

its denotators are representable in the space $Pitch \times Pitch$. This also works in cases where the factors of a **Limit** are not directly **Simple**.

Colimit Denotators

Colimits, disjunctions or coproducts, are again representable in the product space of their cofactors, even if they then typically do not have defined values in all of the product's dimensions. For "missing" dimensions, we set standard values, so that the denotators are represented on a hyperplane in the entire product space. The case where cofactors share common subspaces is especially interesting, since these subspaces will always be populated.

An example will clarify this: the *EulerScore* form defined in Part I consists of *EulerNotes* and *Rests*, which share the **Simple** forms *Onset* and *Duration*. The product space of all cofactor spaces is $Onset \times EulerPitch \times Loudness \times Duration$. While $EulerNotes$ fill the entire space, *Rests* are simply represented on the $Onset \times Duration$ plane. Thus, even though $EulerNotes$ and *Rests* are actually separated by a coproduct, both can be shown in the same space.

Power and List Denotators

Power and **List** forms define sets and ordered sets of distinct objects on any hierarchical level. In practice we typically encounter them on the topmost level as for instance with all the forms supported by early *BigBang*, *Score*, $MacroScore$, and

SoundScore. However, it is also conceivable that they occur only in lower levels, such as in Mariana Montiel's more detailed score form, which is defined as[21]

$$Score' : .\mathbf{Limit}(BibInfo, Signatures, Tempi, Lines, GeneralNotes, GroupArticulations, Dynamics).$$

There, **Power**s appear in almost all the coordinator forms, but not on the top level. In this case we can for instance see all *BarLine*, or *Slur* denotators that appear on lower levels as indirect satellites of our main $Score'$.

Power denotators can always be represented as a set of points in the space of their coordinate. An *EulerScore*, for instance, can be shown as a cloud of objects in the *EulerNoteOrRest* space described in above. **List** denotators can be shown the same way, however, at the expense of the order of their elements, for it may contradict the spacial organization. In any case, **Power** and **List** forms are in fact the main constructs that define the discrimination of distinct visual objects. Wherever they occur, we have the opportunity to define as many elements as we would like.

6.3.2 From a General View Concept to BigBangObjects

From these characteristics we can imply that all we need to have for a representation of any denotators is a conjunction of the **Simple** spaces and a visualization of clouds of objects within them. These objects can be represented in just the same way as the ones in the generalized piano roll described above, as multidimensional rectangles. Whenever a denotator enters *BigBang*, the visualization space is reset based on its form, and users have the possibility to select any form space and start drawing objects,

[21] Mariana Montiel Hernandez. "El Denotador: Su Estructura, construcción y Papel en la Teoría Matemática de la Musica". MA thesis. Mexico City: UNAM, 1999.

as will be described below.

We arrive at the core part of our generalization. In short, the representation of any arbitrary denotator relies on the fundamental difference between the various types of compound forms, **Limit**, **Colimit**, and **Power**. We propose a novel system of classification that generalizes the previous notion of anchors and satellites, based on occurrences of **Power** denotators. For this, we maintain the following rules:

Rule 2 The general visualization space consists of the cartesian product of all **Simple** forms spaces appearing anywhere in the anatomy of the given form. For instance, if we obtain a $MacroScore$ denotator of any hierarchical depth, this is $Onset \times Pitch \times Loudness \times Duration \times Voice$.

Rule 3 Any **Simple** form X the module of which has dimension $n > 1$ is broken up into its one-dimensional factors $X_1, \ldots, X_n$. The visual axes are named after the dimension they represent, i.e. X_n, or X if $n = 1$.

Rule 4 If the same **Simple** form occurs several times in a **Limit**, it is taken to occur several times in the product as well. For instance, the product space of $Dyad$ is $Pitch \times Pitch$. However, if the same **Simple** form occurs at different positions in a **Colimit**, this is not the case. For instance, **Colimit** of $Pitch$ and $Pitch$ results in the space $Pitch$. This renders the space more simple, but we also lose some information. This loss can be regained thanks to an additional spacial dimension, *cofactor index*, as described under Rule 7.

Rule 5 **Power** or **List** denotators anywhere in the anatomy define an instantiation of distinct visual objects represented in the conjoined space. Objects at a deeper level, i.e. contained in a subordinate **Power** or **List**, are considered *satellites* of the higher-level object and their relationship is visually represented by a con-

necting line. For example, *SoundScore* objects formerly considered modulators are now visually no different from regular satellites.

Rule 6 Given a view configuration, the only displayed objects are denotators that contain *at least one* **Simple** form currently associated with one of the visual axes. However, if an object is a satellite and one of the **Simple** forms associated with the axes occurs anywhere in its parental hierarchy, it is represented at exactly that value.

Rule 7 If there is an occurrence of either **Colimit**s or *satellites*, additional dimensions are added to the ones defined in Rules 1-3. For **Colimit** we add *cofactor index*, and for satellites *sibling index* and *satellite level*.[22] These dimensions are calculated for each object and can be visualized in the same manner as the other ones. For instance, associating satellite level with y-position facilitates the selection of all denotators on distinct positions of the satellite hierarchy.

We call the objects defined by these rules `BigBangObjects`. They are not only visual entities, but they are the entities that the *BigBang* rubette deals with in every respect. All operations and transformations available in *BigBang* are applied to sets of `BigBangObjects`, as we will see in the next chapter. The consequence is that we simplify the structure of forms and denotators significantly, so that if we, for instance, are handling denotators of a form defined as **Limit** of **Limit** of **Limit** and so on, we can treat it as a single object. New objects are broken up only if there are **Power**s or **List**s in the hierarchy. We can thus for instance claim that in *BigBang*, we assume that

$$\mathbf{Limit}(A, \mathbf{Limit}(B, \mathbf{Limit}(C, D))) = \mathbf{Limit}(A, B, C, D).$$

[22] Already present in the early *BigBang*, as seen in Section 6.2.1.

Implications for Satellites

One of the main innovations of these definitions is a new notion of the concept of satellites. Previously, the term was uniquely used to describe *Notes* in a *MacroScore* that are hierarchically dependent on other *Notes*. For instance, the analogous construction of *Modulators* in *SoundScores* was not referred to in this way, neither was the relationship represented the same way as satellites are.[23] Following Rule 5 above, *Modulators* are now equally considered satellites and represented in precisely the same way. Another new aspect of this is that now satellites do technically not have to have a shared space with their anchor. An instance, if we define

$$HarmonicSpectrum : .\mathbf{Limit}(Pitch, Overtones),$$
$$Overtones : .\mathbf{Power}(Overtone),$$
$$Overtone : .\mathbf{Limit}(OvertoneIndex, Loudness),$$
$$OvertoneIndex : .\mathbf{Simple}(\mathbb{Z}),$$

Overtones do not have a *Pitch* themselves, but merely a *Loudness*. Because of Rule 6, however, if we choose to see $Loudness \times Pitch$ as the axes of a view configuration, the *Overtones* are represented above their anchor. An example visualization of this form will be shown below.

Above, we discussed how satellites and modulators were defined relatively to their anchors (Section 6.2). This can also be generalized for the new notion of satellites. We add another rule:

Rule 8 Given a **Simple** form F, every denotator $d_i : @F$ in a satellite `BigBangObject`, i being its index in case the satellite contains several denotators of form F, is

[23] Thalmann and Mazzola, "Gestural Shaping and Transformation in a Universal Space of Structure and Sound".

defined in a relative way to $d_i@F$ in its anchor, if there is such a denotator.

For instance, if we define

$$MacroDyad : .\mathbf{Power}(DyadNode),$$
$$DyadNode : .\mathbf{Limit}(Dyad, MacroDyad),$$

the first *Pitch* in each satellite *Dyad* is defined relatively to the first *Pitch* in its anchor, and the second *Pitch* in each satellite relatively to the second *Pitch* in the anchor. On the other hand, in a *HarmonicSpectrum* none of the satellites share **Simple** denotators with their anchor and are thus defined absolutely.

6.3.3 New Visual Dimensions

The facts view of the new *BigBang* maintains all the features of the early *BigBang* and can still be navigated the same way as described in Section 6.2.2. However, the newest version allows independent zooming in and out horizontally and vertically, when the shift or alt keys are pressed. It also features some additional view parameters. There is now an option to, instead of hue (*Color*) values, use RGB values for color, in a similar way as in *PrimaVista*. This adds more visual variety at the expense of the cyclical nature of hue, and is especially beneficial when working with data types other than musical ones, such as images. The new view parameters vector thus looks as follows:

$$N' = \{X\text{-}Position, Y\text{-}Position, Width, Height, Alpha, Red, Green, Blue\}$$

In the future, more visual characteristics can easily be added, such as varying shapes, texture, or a third dimension.

The former note parameters, in turn, now called *denotator parameters*, include $SiblingNumber$ and $ColimitIndex$ where appropriate and vary according to the input or chosen form. The former identifies the index of a denotator in its **Power** or **List**, whereas the latter refers to an index based on all possible combinations of **Colimit** coordinates. For instance, for an object form

$$\mathbf{Colimit}(\mathbf{Colimit}(X_0, X_1), \mathbf{Colimit}(X_2, X_3, X_4), X_5),$$

where $X_0, ..., X_5$ are any other forms not containing **Colimit**s, we get six possible configurations: an object containing X_0 gets index 0, one containing X_1 gets 1, and so on.

For $EulerScore$ denotators, for instance, the entire denotator parameters look as follows:

$$M_{EulerScore} = \{Onset, EulerPitch1, EulerPitch2, EulerPitch3,$$
$$Loudness, Duration, ColimitIndex\}.$$

The three-dimensional space of $EulerPitch$ is broken up into its three constituate dimensions, and $ColimitIndex$ is added, with two potential values: 0 for $EulerNoteOrRests$ containing an $EulerNote$, 1 for $EulerNoteOrRests$ with a $Rest$.

6.4 The Sonification of BigBangObjects

As seen above in Section 6.2.3, in the early *BigBang* rubette, sonification was relatively straightforward, since all objects that had to be dealt with existed in the same five-dimensional space. For the new *BigBang*, this concept had to be generalized as well. For users to be able to sonify a multitude of denotators, even ones they define

themselves, the sonification system had to become more modular.

Our solution generalizes the `JSynNotes` described above into `JSynObjects`, which can contain any number of a set of standard musical parameters. Each `BigBangObject` is converted into a `JSynObject`, by searching for occurrences of these musical parameters anywhere in their anatomy. Any parameters necessary for a sounding result subsequently obtain a standard value. For instance, if we play back a **Simple** denotator *Pitch*, a `JSynObject` is created with a standard *Loudness*, *Onset*, and *Duration*, so that it is audible. Especially *Onset* and *Duration* are relevant in this case. The standard values assigned for temporal parameters are chosen such that the object plays continuously for as long as the denotator is being played. This is particularly interesting when the denotator is transformed, which results in continuously sounding microtonal sweeping.

`JSynObjects` can also have multiple pitches, in order to work with denotators such as *Dyad*, as defined in Section 6.3.1, or other user-defined types that might describe chords, and so on. Some of the recognized simple forms so far are all note parameters (*Onset*, *Pitch*, *Loudness*, *Duration*, *Voice*), as well as *BeatClass*, *ChromaticPitch*, *PitchClass*, *TriadQuality*, *OvertoneIndex*, *Rate*, *Pan*, and *OperationName*. *Rate* replaces *Onset* by defining the rate at which a `JSynObject` is repeatedly played, *OperationName* distinguishes between frequency modulation, ring modulation, and additive synthesis, and *TriadQuality* adds an appropriate triad above each *Pitch* in the object, assuming that they are root notes. Some of the other forms are discussed below, along with examples of the visualization of their denotators.

Another recent addition is the option of having everything played back through MIDI, either with Java's internal MIDI player, or by sending live MIDI data to any other application or device, via IAC bus or MIDI interface. MIDI is event-based and thus problematic for playing continuous objects without temporal parameters. There

are two solutions to this problem implemented so far. Either, objects are repeated continuously at a specified rate, or a note off event is only sent when an object is replaced by another. In the latter case, note ons are only sent again once a denotator is transformed.

In order to play back the composition in *BigBang*, users can press the play button in the lower toolbar. If the denotator has a temporal existence, i.e. it contains *Onsets*, it can be looped, where the player automatically determines the loop size to be the entire composition. In addition to this, any musical denotator in *BigBang* can be played back by using external MIDI controller such as a keyboard controller. Each MIDI key of such a controller triggers one performance of the denotator, i.e. a one-shot temporal playback, a loop, or a continuous playback, depending on the denotator. Middle C (60) corresponds to the visible denotator, while all other keys trigger transposed versions, e.g. a half step up for 61, etc. This is especially practical when designing sounds, i.e. denotators without *Onset* or *Duration*, such as the *HarmonicSpectrum* form defined above. This way, users can design sounds and immediately play the keyboard with them, just as with a regular synthesizer.

For the future, this system of sonification could be extended in order to work in a similar way to view configurations. For now, whenever a new **Simple** form is introduced that should be sonified in a novel way, the system has to be adjusted accordingly. With a free association of any **Simple** form with a sonic parameter, just as it is done for the visual system, users can experiment with spontaneously performing parameter exchanges, or with sonifying non-musical forms.

6.5 Examples of Forms and the Visualization of their Denotators

In this chapter, we have discussed what the objects on *BigBang*'s factual level are and how they are visualized and sonified. It is now time to give some specific examples of forms that can potentially be defined and show how their denotators are visualized. Sonification will have to be left to the readers to try themselves. Anything we feed the new BigBang rubette will be analyzed and visualized as described above. Users may also select a form within *BigBang* upon which the facts view is cleared and they may simply start drawing denotators, as will be described in the next chapter.

I will start with some simple constructs from set theory, move to tonal constructs, and finally give some examples from computer music and sound design.

6.5.1 Some Set-Theoretical Structures

The most basic construct to be represented is necessarily a single **Simple** denotator. For instance, if we input a *Pitch*, the space is merely one-dimensional, but it can be represented in various visual dimensions simultaneously. Figure 6.7 shows the pitch $middleC$: $@Pitch(60)$ – C4 is MIDI pitch 60 – being represented in every possible visual dimension in RBG mode,[24] however reasonable this may be. X, y, width, height, alpha, red, green, and blue, all represent the value 60, depending on the min_m, max_m defined (see Section 6.2.1).

For a **Power** of a **Simple**, we already get a cloud of values. Figure 6.8 shows an

[24]Explained in Section 6.3.3)

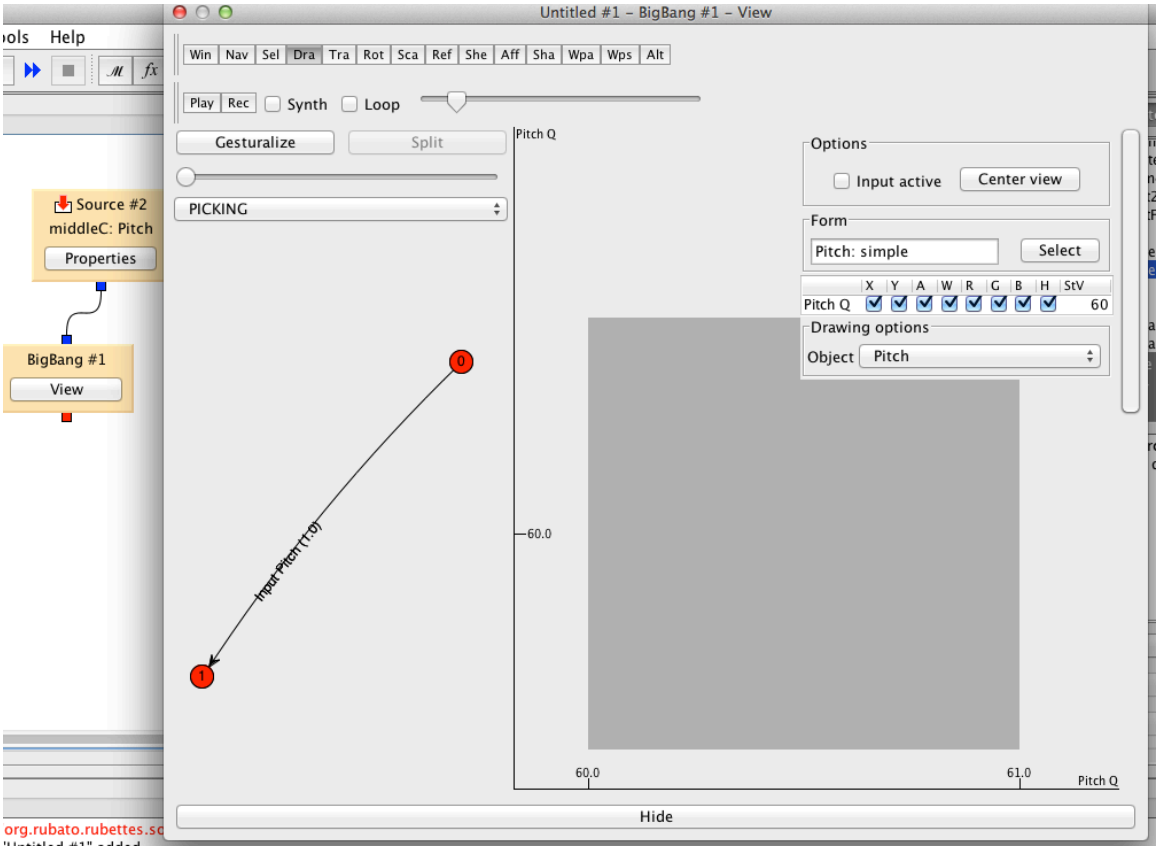


Figure 6.7: The new *BigBang* rubette visualizing *Pitch* denotator in every visual dimension.

example of a

$$PitchSet : .\mathbf{Power}(ChromaticPitch),$$

$$ChromaticPitch : .\mathbf{Simple}(\mathbb{Z}).$$

Note that *ChromaticPitch* differs from *Pitch* in that it only allows for integer values, which models the Western equal-tempered chromatic pitch space. In the figure, *ChromaticPitch* is shown on both axes, color, width and height. This way, we can define all sorts of datatypes commonly used in music theory or sound synthesis and visualize and sonify them. If we wanted, for instance, to compose with pitch

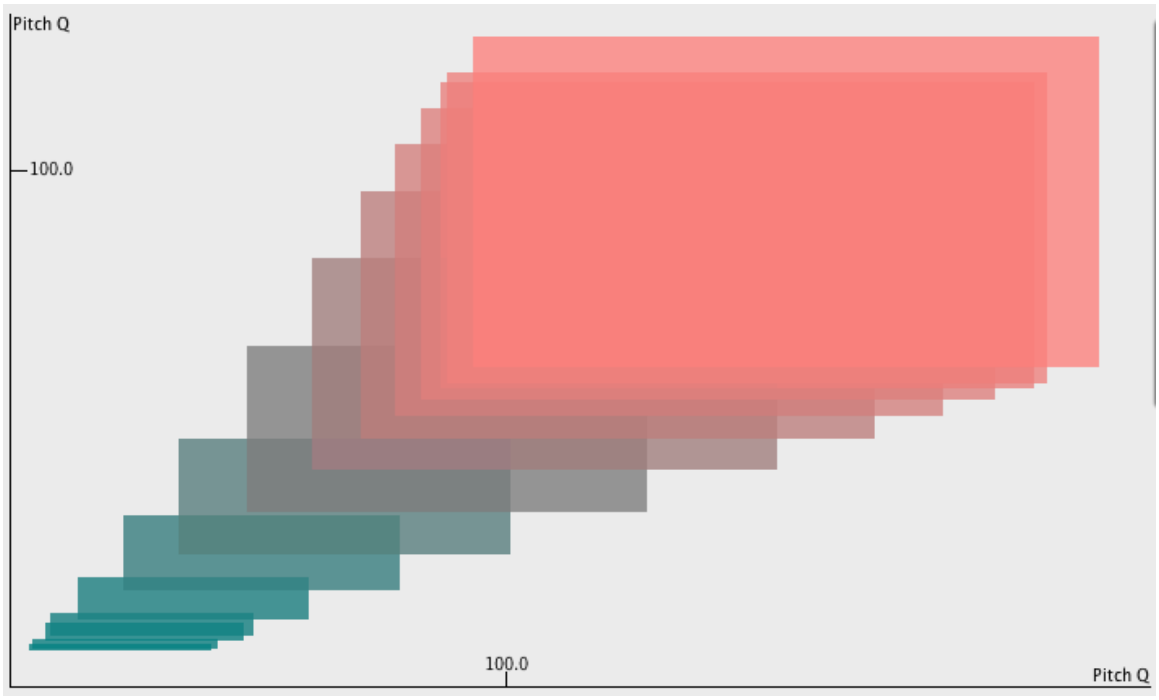


Figure 6.8: A *PitchSet* simultaneously visualized using several visual characteristics.

classes instead of pitches, we could define

$$PitchClassSet : .\mathbf{Power}(PitchClass),$$
$$PitchClass : .Simple(\mathbb{Z}_{12})$$

If we wish to work with pitch-class trichords, a common construct in set theory, we can define

$$Trichords : .\mathbf{Power}(Trichord),$$
$$Trichord : .\mathbf{Limit}(PitchClass, PitchClass, PitchClass).$$

PitchSets and *PitchClassSets* can also be realized as ordered sets. We simply

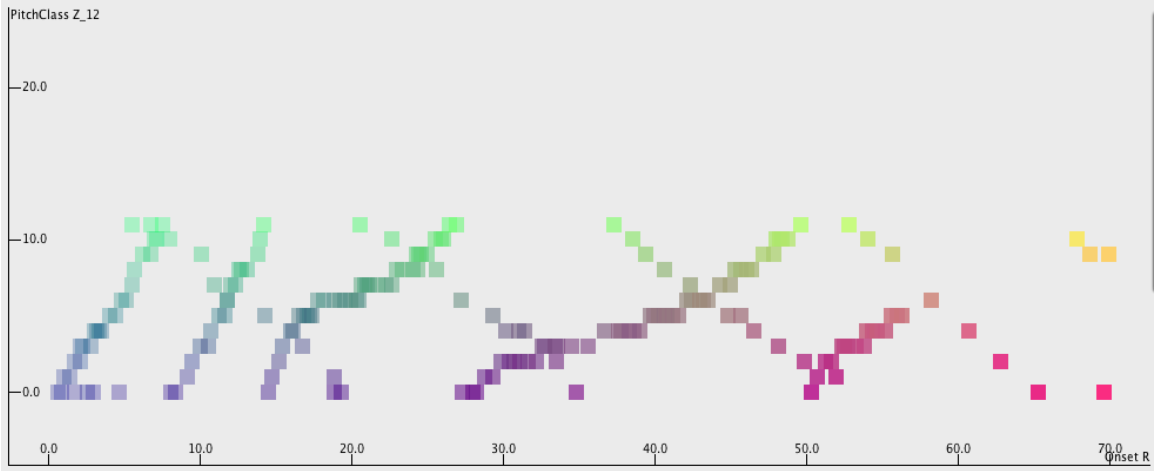


Figure 6.9: A *PitchClassScore* drawn with ascending and descending lines to show the cyclicality of the space.

need to replace **Power** with **List**, e.g.

$$OrderedPitchSet : .\mathbf{List}(Pitch).$$

In order to compose with *PitchClasses* the same way we can compose with *Scores*, i.e. create temporal structures, we can define

$$PitchClassScore : .\mathbf{Power}(PitchClassNote),$$

$$PitchClassNote : .\mathbf{Limit}(Onset, PitchClass, Loudness, Duration, Voice)$$

which is then visualized as shown in Figure 6.9.

6.5.2 Tonal and Transformational Theory

The next few examples imitate spaces and constructs from transformational theory and traditional music theory.[25] For a model of triads, as they are often used in

[25]Some of them were described in Thalmann and Mazzola, "Visualization and Transformation in a General Musical and Music-Theoretical Spaces".

transformational theory, we define

$$Triad : .\mathbf{Power}(Pitch, TriadQuality),$$
$$TriadQuality : .\mathbf{Simple}(\mathbb{Z}_4),$$

where Quality stands for one of the four standard qualities in tonal music: diminished, minor, major, and augmented.

More generally, a simplified notion of chord progressions can be implemented as follows

$$Progression : .\mathbf{List}(Chord),$$
$$Chord : .\mathbf{Limit}(Onset, PitchSet, Loudness, Duration),$$

assuming that all members of a chord have the same temporal and dynamic qualities. In so doing, the pitches of a chord are actually satellites and thus also visualized this way, as can be seen in Figure 6.10. From there, we can also define hierarchical chord progressions the same way as we did this above for *Score* or *Dyad*. For instance, we can define

$$MacroProgression : .\mathbf{List}(ChordNode),$$
$$ChordNode : .\mathbf{Limit}(Chord, MacroProgression),$$

This way, each chord can have ornamental progressions, just as we know it from Schenkerian theory. If a main progression is transposed, its ornamental progressions, defined in a relative way to them, are transposed with it. The next chapter will clarify what this means.

Figure 6.11 shows an example of the depiction of **Colimit**s. It shows a denotator

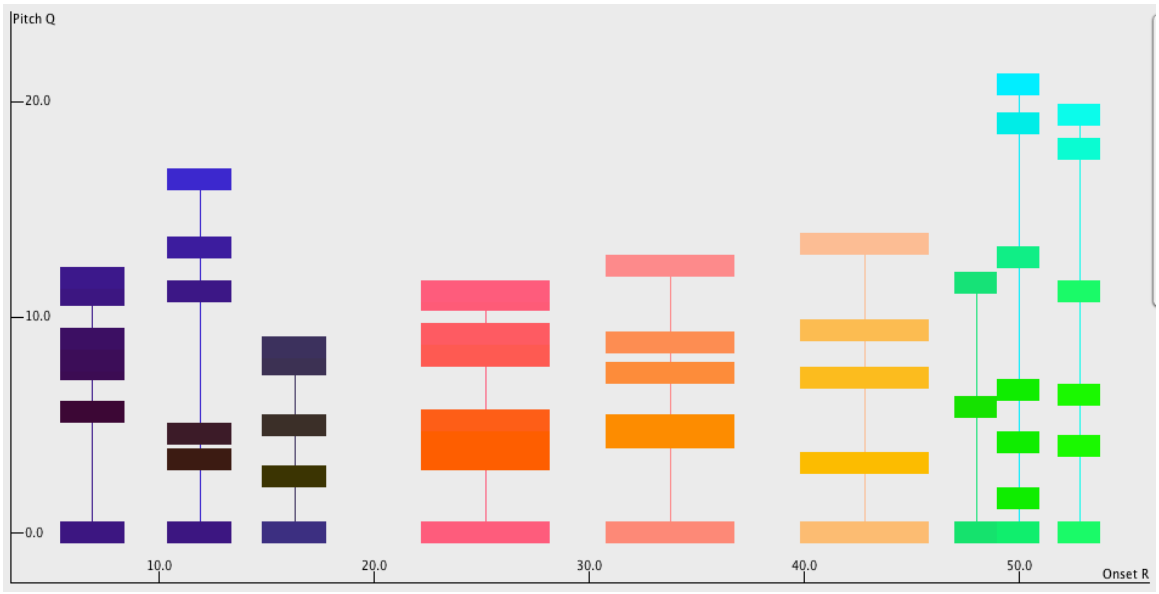


Figure 6.10: A *Progression* where pitches adopt the visual characteristics of their anchor chord.

of a form similar to *EulerScore*, but with regular *Pitch* and an additional *Voice* parameter, thus simply using regular *Notes* and *Rests*. In the image we see that all the rests are depicted at *Pitch* 0, since they do not contain a *Pitch*. If we chose to depict the denotator on the *Onset* × *Duration* plane, the rests would also be shown in two dimensions.

A final example illustrates a way we can introduce rhythmical relationships other than using *Onset*. If we write

$$\begin{aligned} &\mathit{Texture} : .\mathbf{Power}(\mathit{RepeatedNote}), \\ &\mathit{RepeatedNote} : .\mathbf{Limit}(\mathit{Pitch}, \mathit{Loudness}, \mathit{Rate}, \mathit{Duration}), \\ &\mathit{Rate} : .\mathbf{Simple}(\mathbb{R}) \end{aligned}$$

we obtain a set of notes that are repeated at a certain rate, altogether forming a characteristic *Texture*.

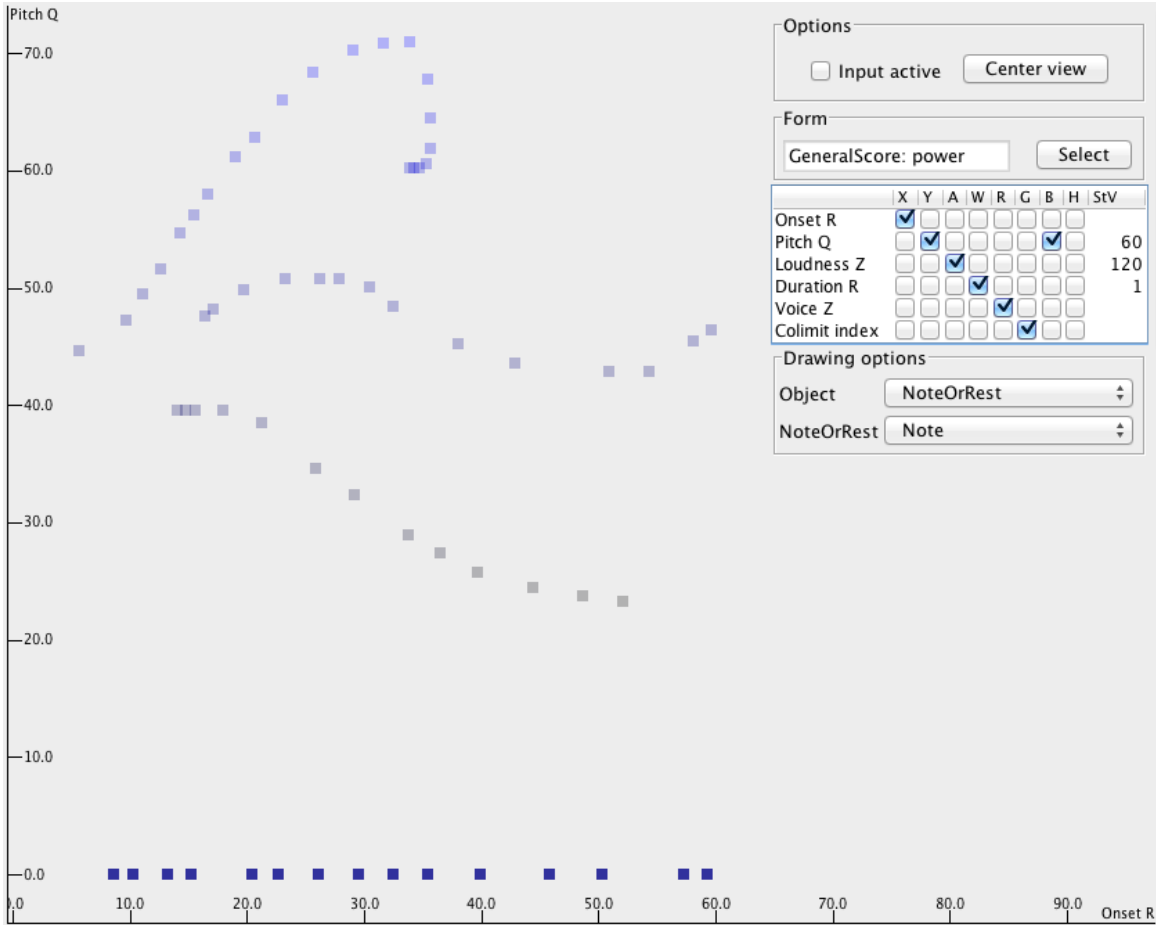


Figure 6.11: A *GeneralScore* with some *Notes* and *Rests* shown on the $Onset \times Pitch$ plane.

6.5.3 Synthesizers and Sound Design

Finally, here are some examples of forms that allow for more sound- and timbre-oriented structures. Some of the forms shown in Section 6.5.1 could already be considered to be sound-based forms as they may be seen as somewhat related to additive synthesis, but we can go much farther than that.[26]

[26]Some of these constructions were described in Thalmann and Mazzola, "Using the Creative Process for Sound Design based on Generic Sound Forms".

For instance, we can define

$$Spectrum : .\mathbf{Power}(Partial),$$

$$Partial : .\mathbf{Limit}(Loudness, Pitch).$$

This models a constantly sounding cluster based on only two dimensions. Since it is not using *ChromaticPitch* but *Pitch*, the cluster can include any microtonal pitches. Figure 6.12 shows an example of a *Spectrum*. If we, however, wanted to define a spectrum that only allows for harmonic overtones, this form would not be well-suited, as we would have to meticulously arrange each individual pitch so that it sits at a multiple of a base frequency. Instead, we could simply use the form already introduced above, *HarmonicSpectrum* (Section 6.3.2). Figure 6.13 shows an example denotator of a set of harmonic spectra, defined as $HarmonicSpectra$: .**Power**($HarmonicSpectrum$). Since satellites (*Overtones*) and anchors ($HarmonicSpectrum$) do not share **Simple** dimensions, they can only be visualized if one **Simple** of each is selected as an axis parameter, here $Pitch \times OvertoneIndex$. However, as we will see in the next chapter, they can both be transformed in arbitrary ways on such a plane. These are examples of the simplest way of working with additive synthesis in *BigBang*. All oscillators are expected to be based on the same wave form and a phase parameter is left out for simplicity. This is also the case for the following examples.

Even though the previous form leads to more structured and visually appealing results, we limited ourselves to purely harmonic sounds, since all *Overtones* are assumed to be based on the same base frequency *Pitch*. To make it more interesting, we can decide to unite the sound possibilities of *SoundSpectrum* with the visual and structural advantages of *HarmonicSpectrum* by giving each *Overtone* its own *Pitch*.

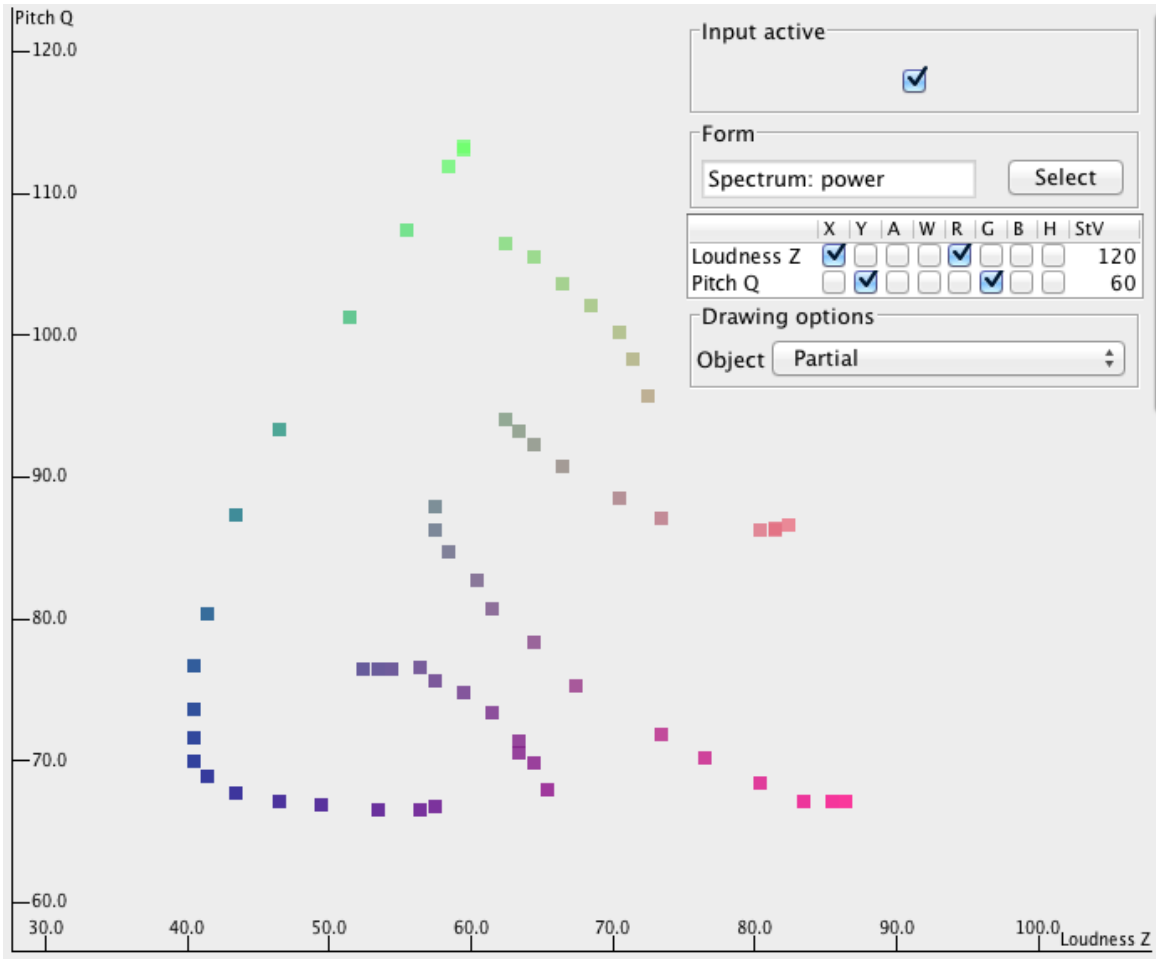


Figure 6.12: A *Spectrum* shown on *Loudness* × *Pitch*.

The following definition does the trick:

$$\begin{aligned} &\mathit{DetunableSpectrum} : .\mathbf{Limit}(\mathit{Pitch}, \mathit{Overtones}), \\ &\mathit{Overtones} : .\mathbf{Power}(\mathit{Overtone}), \\ &\mathit{Overtone} : .\mathbf{Limit}(\mathit{Pitch}, \mathit{OvertoneIndex}, \mathit{Loudness}). \end{aligned}$$

Since values reoccurring in satellites are defined in a relative way to the corresponding ones of their anchor, we get the opportunity to define *deviations* in frequency from the harmonic overtone, rather than the frequencies themselves. A displacement of

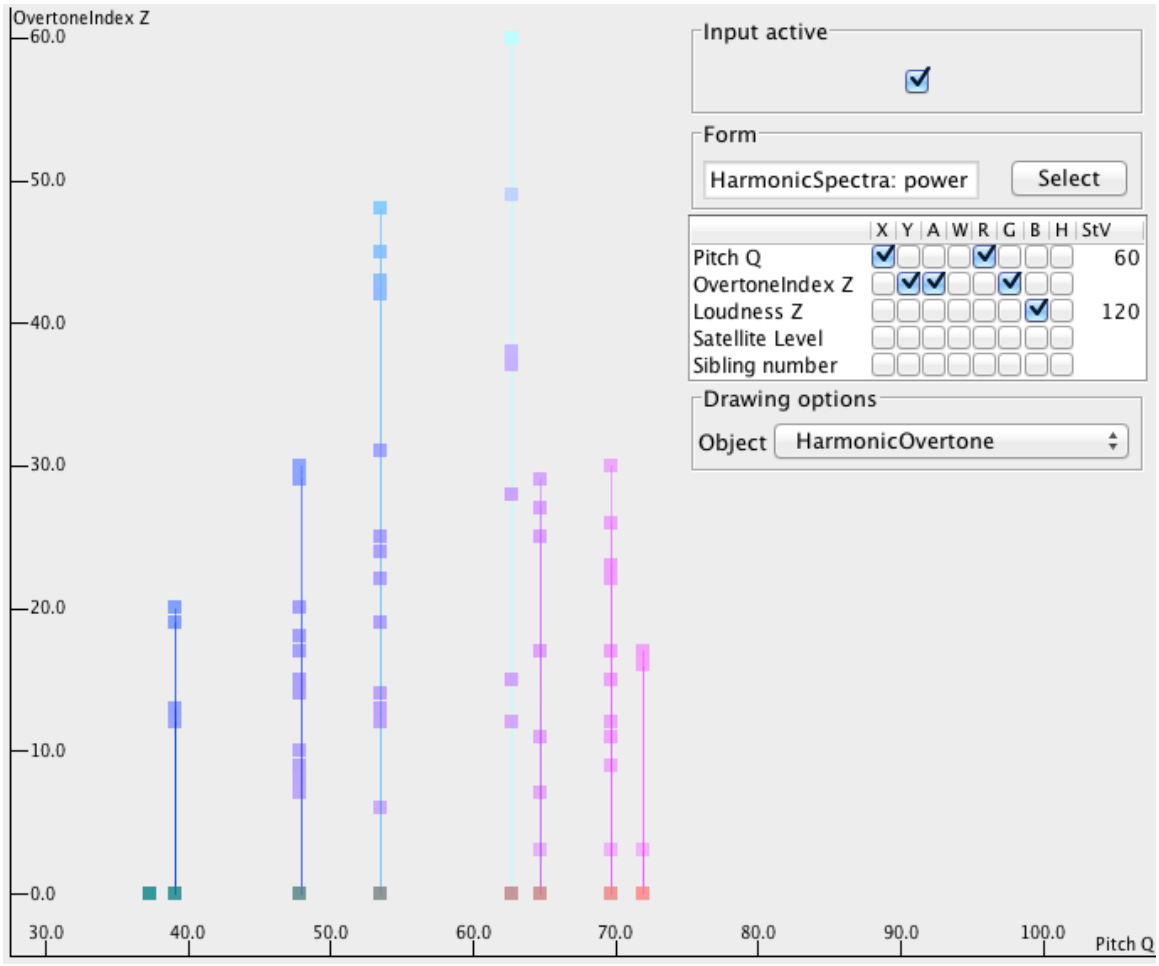


Figure 6.13: A constellation of eight *HarmonicSpectra* with different fundamental *Pitches* and *Overtones*.

a satellite on the *Pitch* axis with respect to its anchor enables us to detune them. Figure 6.14 shows an instance of such a *DetunableSpectrum*.

The three forms above are just a few examples of an infinite number of possible forms. Already slight variants of the above forms can lead to significant differences in the way sounds can be designed. For instance, generating complex sounds with the above forms can be tedious as there are many possibilities to control the individual structural parts. A well-known method to achieve more complex sounds with much fewer elements (oscillators) is frequency modulation, which can be defined as follows

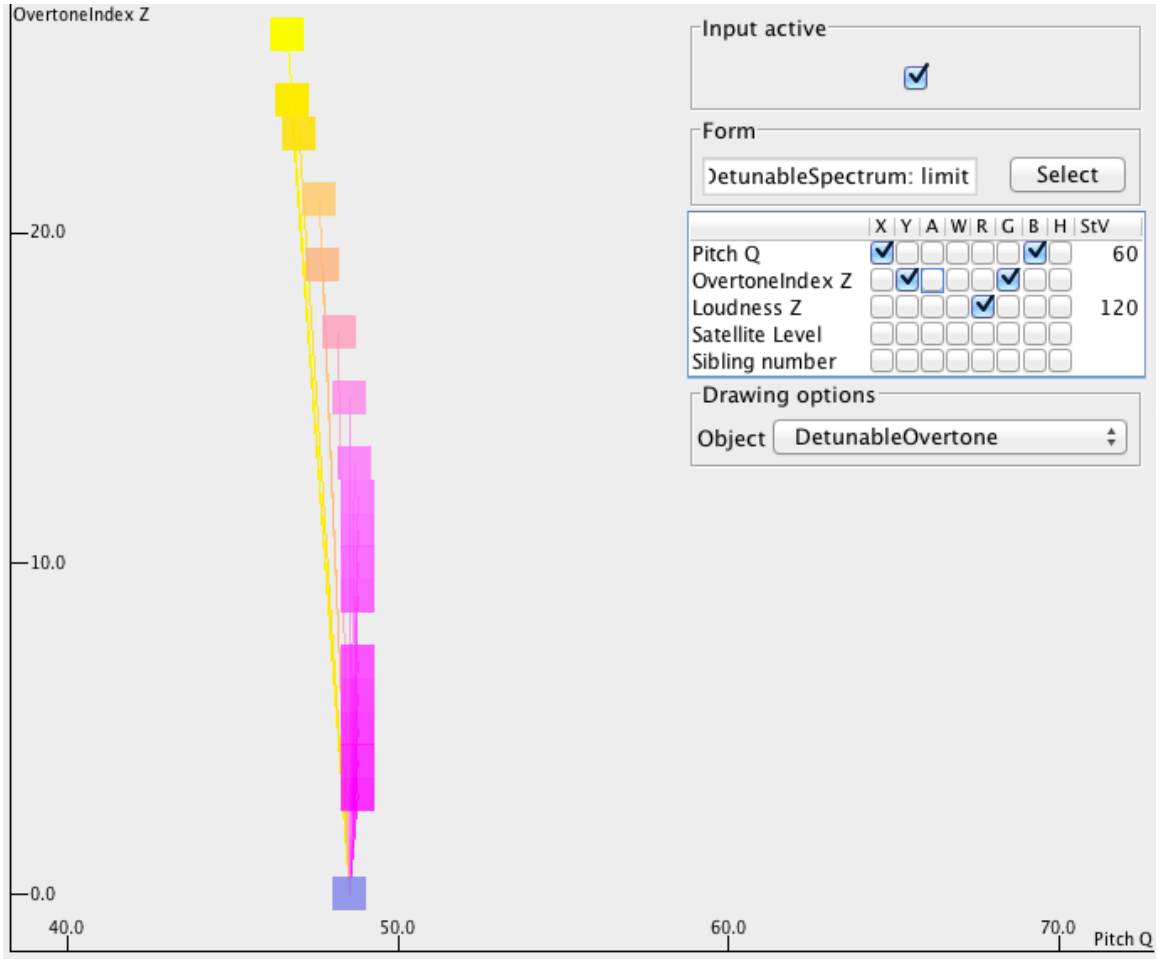


Figure 6.14: An instance of a *DetunableSpectrum*, where the fundamentals of the *Overtones* are slightly detuned.

in a recursive way:

$$FMSet : .\mathbf{Power}(FMNode),$$

$$FMNode : .\mathbf{Limit}(Partial, FMSet),$$

with *Partial* as defined above. Examples as complex as the one shown in Figure 6.15 can be created this way. Frequency modulation, typically considered highly unintuitive in terms of the relationship of structure and sound,[27] can be better understood

[27] John Chowning. "The Synthesis of Complex Audio Spectra by Means of Frequency Modulation".

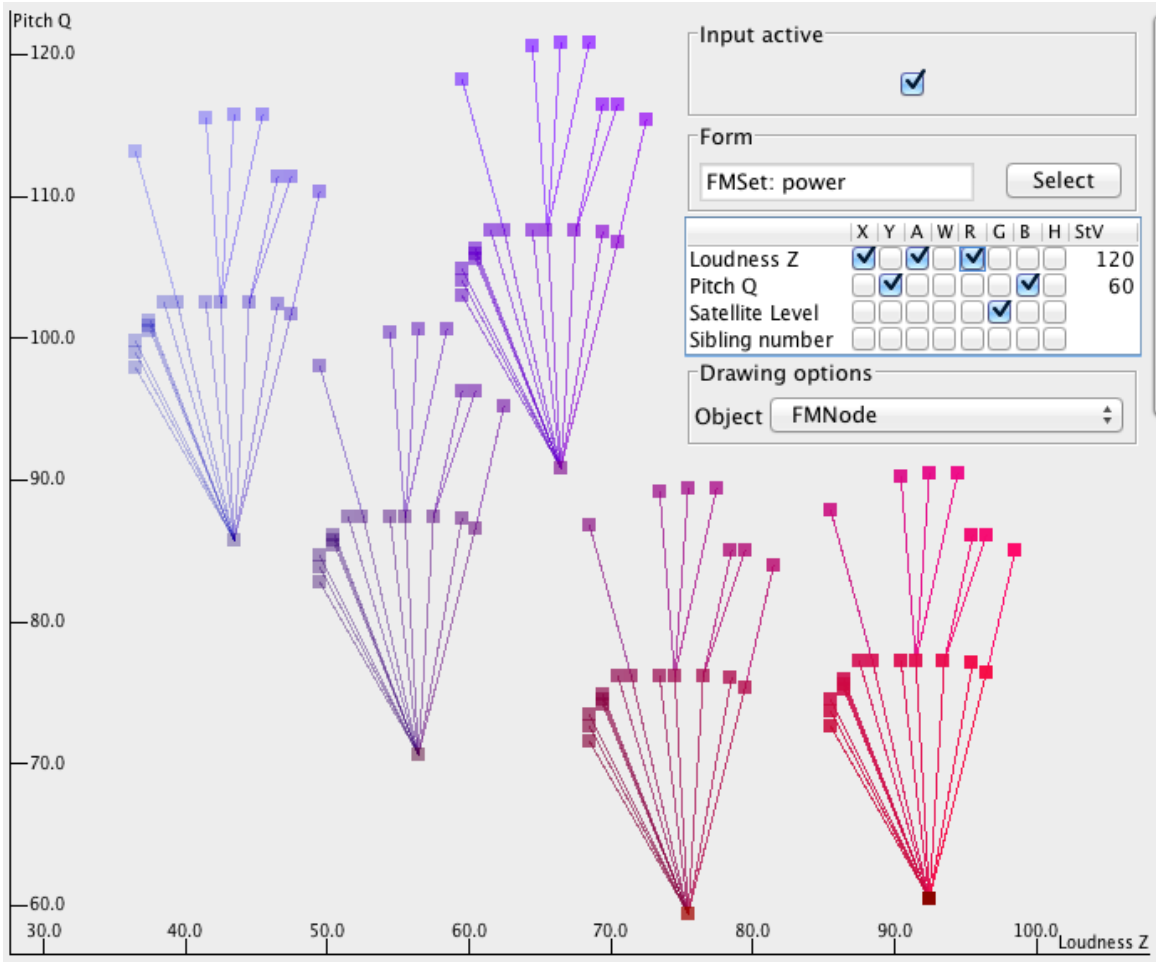


Figure 6.15: An *FMSet* containing five carriers all having the same modulator arrangement, but transposed in *Pitch* and *Loudness*.

with a visual representation such as this one. All carriers and modulators are shown respective to their frequency and amplitude and can be transformed simultaneously and parallelly, which has great advantages for sound design compared to old-fashioned skeuomorphic synthesizers and applications (See Section 4.3.3).

In: *Journal of the Audio Engineering Society* 21 (1973).

In order to include other synthesis models, we can define

$$
\begin{aligned}
&\mathit{GenericSound} : .\mathbf{Limit}(\mathit{Oscillator}, \mathit{Satellites}, \mathit{Operation}),\\
&\mathit{Oscillator} : .\mathbf{Limit}(\mathit{Loudness}, \mathit{Pitch}, \mathit{Waveform}),\\
&\mathit{Satellites} : .\mathbf{List}(\mathit{GenericSound}),\\
&\mathit{Operation} : .\mathbf{Simple}(\mathbb{Z}_3),\\
&\mathit{Waveform} : .\mathbf{Simple}(\mathbb{Z}_4),
\end{aligned}
$$

where *Operation* represents the three synthesis operations for additive synthesis, ring modulation, and frequency modulation. For each anchor/satellite relationship, we can choose a different operation. Each *Oscillator* also has its own *Waveform*, here a selection of four varying ones, for instance sine, triangle, square, sawtooth.[28] Sounds designed this way can immediately be played with by using a keyboard controller, as seen in Section 6.4.

Finally, we can also combine multiple forms into higher-level forms that contain several objects. For instance, a **Limit** of *SoundSpectrum* and *Score* allows us to create compositions containing both constantly sounding pitches and notes with *Onsets* and *Durations*. We simply need to define

$$\mathit{SpectrumAndScore} : .\mathbf{Limit}(\mathit{Spectrum}, \mathit{Score})$$

Figure 6.16 shows an example of such a composition. This way, any number of synthesis methods and musical formats can be combined to higher-level forms and can be used simultaneously in *BigBang*.

These examples show how much structural variety we can create by just using a

[28] A slightly different *GenericSound* form is described in Thalmann and Mazzola, "Visualization and Transformation in a General Musical and Music-Theoretical Spaces".

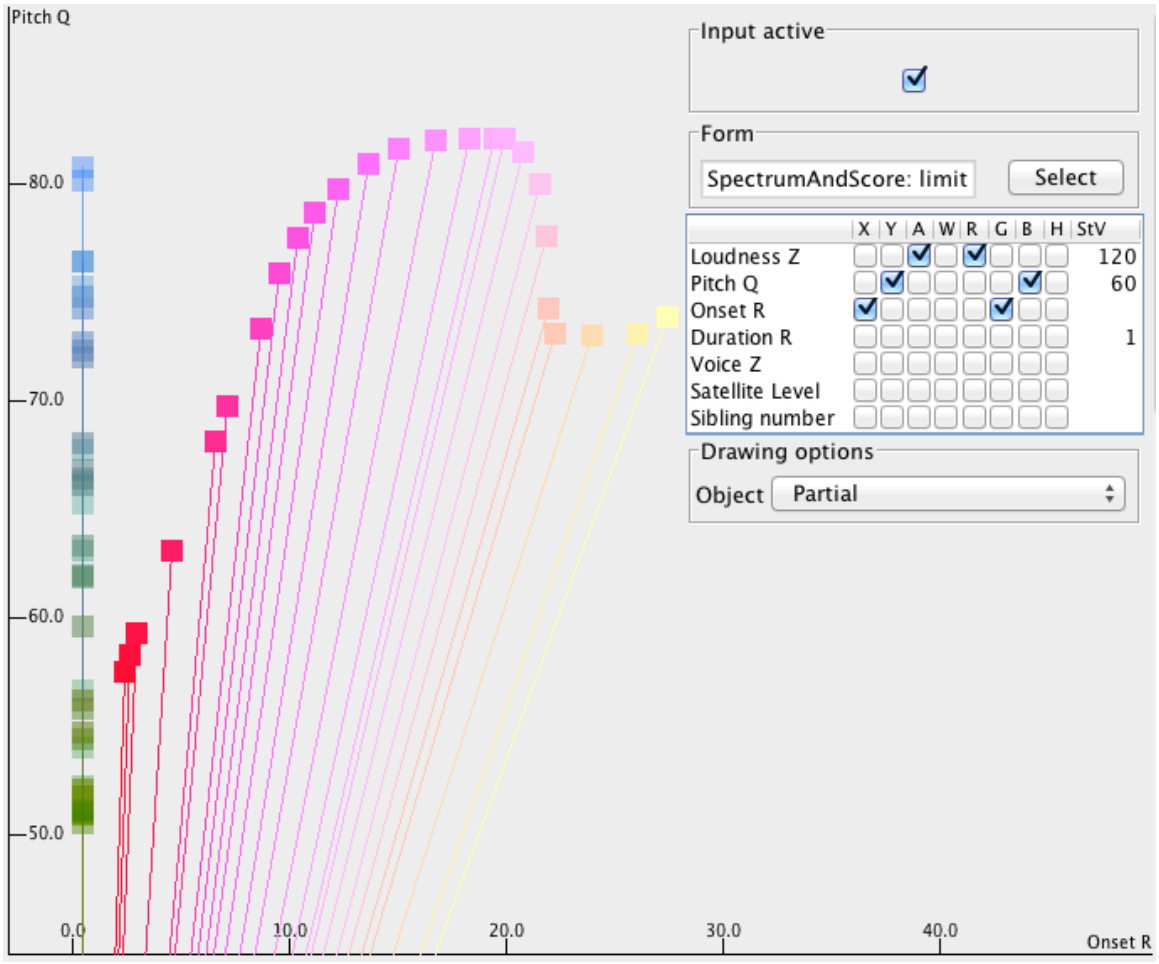


Figure 6.16: A composition based on a **Limit** of a *SoundSpectrum* (*Pitches* at *Onset* 0) and a *Score* (*Pitches* with *Onsets*).

small given set of **Simple** forms, and how their visualization can help us understand the structures. All of them can directly be sonified, even while we are building the denotators. Most importantly, such forms can be defined at runtime in *Rubato Composer* and they can immediately be used in *BigBang*. In addition to musical data types, such as the examples here, one can define forms describing any kind of fact. For example, during the work on this thesis I programmed rubettes that read image files (*ImageFileIn*), translate them into forms, and make them available to transformation in *BigBang*, before being, exported again or converted into musical objects by other

rubettes.

In the next chapter I will discuss how such objects, once their form is defined, can be created, manipulated, and transformed in *BigBang*. For this, we need to examine how the *BigBang* rubette implements the level of processes.

Chapter 7

Processes: BigBang's Operation Graph

The main intention behind the *BigBang* rubette is to give composers and improvisers a way to use *Rubato Composer* that is more intuitively understandable, more spontaneous, and more focused on audible results than on the mathematical underpinnings. After discussing the type of facts available in *BigBang* we need to examine how we can make them and what we can do with them. From the first part of this thesis we now know that both of these activities, making and manipulating, are instances of processes. *BigBang* keeps track of these processes in a more sophisticated way than other musical software, especially ones dedicated to symbolic structures.

Rubato Composer itself already allows users to create processes by building networks of rubettes. Why does *BigBang* need its own processes? There are several significant differences between the processes of *Rubato Composer* and other systems such as *Max/MSP*, and the ones of *BigBang*.

1. *BigBang*'s visualization of processes is the dual graph of the rubette networks in *Rubato Composer*. Its focus on transformations as arrows is closer to the

way we imagine processes, as seen in the first part of this thesis. *Rubato Composer*'s representation of denotators traveling through connecting lines imitates the physical reality of electric signals traveling through cables and has hardly anything to do with our imagination and representation of the mathematical constructs, e.g. diagrams of morphisms in category theory, or transformational graph.

2. *BigBang* emphasizes spontaneity and a quick work process. Rather than representing a definite composition process, its processes represent experimental stages, are easily mutable, and allow for the creation of alternatives.

3. Its processes focus on a simple vocabulary of operations and transformations that can be combined to create larger structures in a transparent way.

4. The processes in *BigBang* are always directly connected to facts, and the user has the chance to observe and interact with both, facts and processes, simultaneously while composing or improvising. In *Rubato Composer*, facts remain hidden for large parts of the process.

5. *BigBang* focuses on processes that can be directly connected to gestures. For many of *Rubato Composer*'s features this would be difficult to do.

6. Most importantly, its processes represent the workings of the rubette itself. Users can use it just the way they use Macro Rubettes,[1] which encapsulate entire *Rubato Composer* networks within them. For instance, they can define a compositional process in *BigBang*, duplicate the rubette, and use it in multiple contexts by adjusting the process as needed. Thereby they can even decide to send denotators of any other forms and see what the rubette yields.

[1]See Milmeister, *The Rubato Composer Music Software: Component-Based Implementation of a Functorial Concept Architecture*, p. 237.

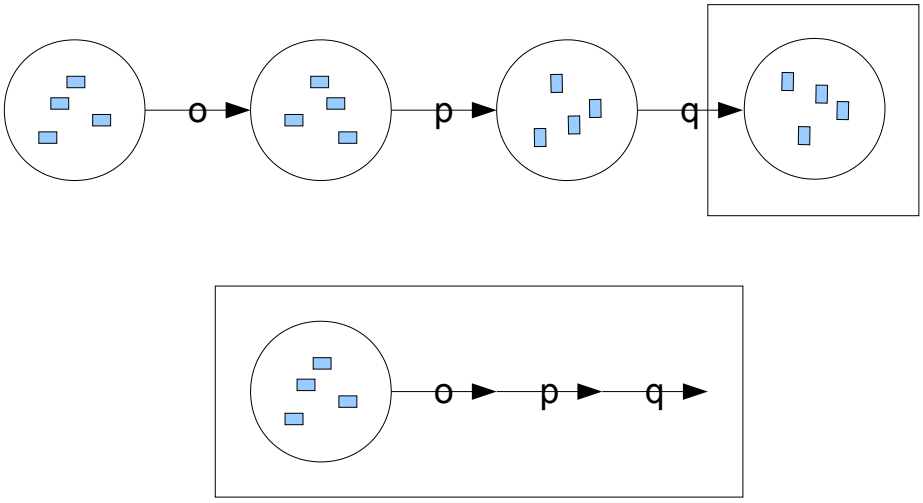


Figure 7.1: A factual notion of a composition above, versus a dynamic notion below.

Despite these differences, there is a chance that someday *Rubato Composer* may be extended to adopt some of *BigBang*'s principles, so that they can be used on a higher level and in a greater variety of ways.

Underlying the representation of processes in *BigBang* is a notion of a composition or improvisation as a dynamic rather than static entity. Rather than seeing the composition as a definite fact, we see it as a conjunction of a set of musical input objects along with a processual graph into which these objects are fed, similar in this respect to transformational theory. Figure 7.1 shows a diagram of this scheme.[2] This way, the only existing facts are the initial facts, such as input denotators, and everything else can be produced by the transformations in the graph. Any stage of the composition process, i.e. any node of the process graph, can be dynamically generated using the factualizing procedure first introduced in Section 2.3. Internally, denotators are never saved at every step of the composition, but always generated dynamically.

[2]These concepts were first discussed in Thalmann and Mazzola, "Poietical Music Scores: Facts, Processes, and Gestures"; Guerino Mazzola and Florian Thalmann. "Musical Composition and Gestural Diagrams". In: *Proceedings of the Third International Conference on Mathematics and Computation in Music (MCM)*. ed. by C. Agon et al. Heidelberg: Springer, 2011.

In this chapter, I will first introduce the available operations and transformations and describe what they do. Then I will describe how factualizing works in *BigBang*. Finally, I will discuss the way processes are visualized and how users can interact with the visualizations.

7.1 Temporal BigBangObjects, Object Selection, and Layers

Before performing an action, we need to be able to decide which objects are going to be affected by the action we wish to perform. In *BigBang*, users can make selections in the facts view by drawing rectangles around objects. During this process, they can arbitrarily change their perspective. For instance, it may be tedious to select all visible second-level satellites in a complex *MacroScore* composition. By selecting *SatelliteLevel* as one of the visible axes, the composition will be shown as a number of levels (Figure 7.2). The user can then simply draw a rectangle around the objects shown on level two to select all satellites there.

Whenever the user performs an operation, it will be applied to all selected objects. When selecting all objects, the operation or transformation will be applied to as many objects as were selected, even if the input of the rubette changes. For instance, suppose we input a piece with twelve notes into *BigBang*, select all of them, and then transform them. If we then input another composition with many more notes, the transformation will be applied only to the twelve first notes of the new composition.

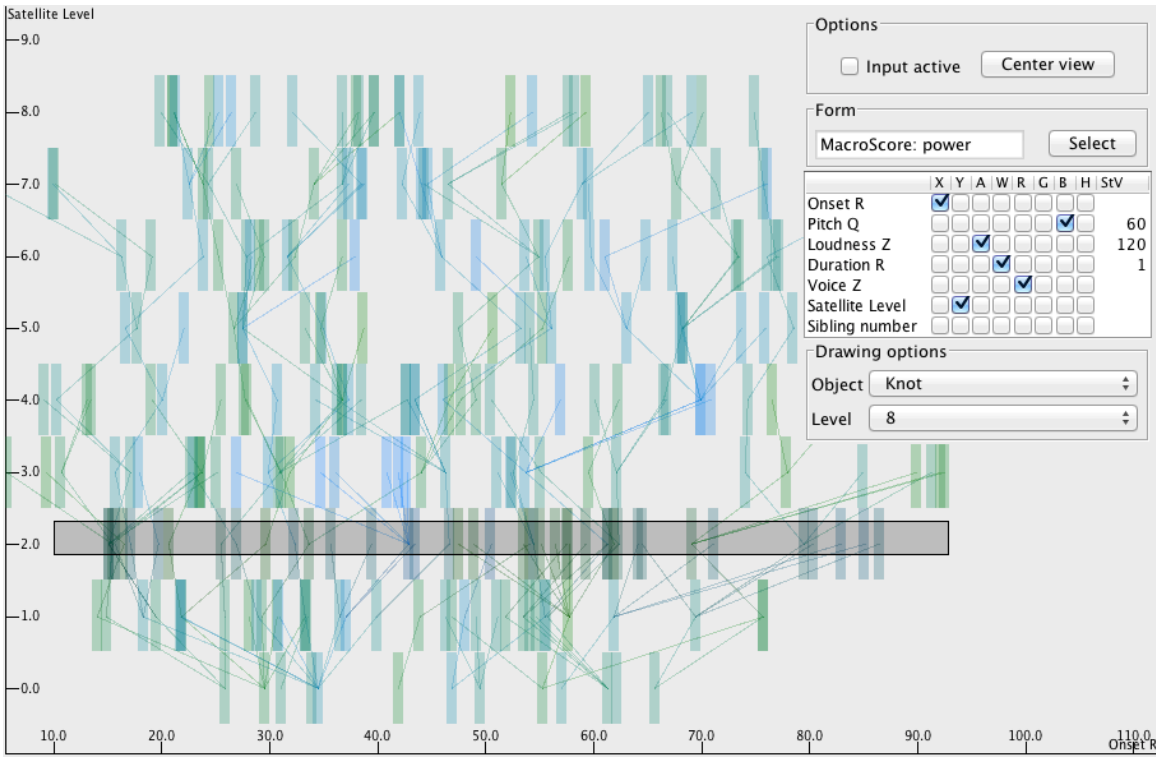


Figure 7.2: A *MacroScore* with its second level of satellites being selected. Note that the y-axis is *SatelliteLevel* to facilitate the selection. The x-position and colors of the objects hint at their chaotic arrangement on the $Onset \times Pitch$ plane.

7.1.1 Selecting None and Lewin's Transformation Graphs

If *none* of the objects are selected, an operation or transformation will always be applied to *all* objects present at the respective state, however many there may be. This has a major advantage, besides speeding up the compositional process of users that like to work with the full set of musical objects. Users may often be in the situation where they know the actions they would like to perform without being sure what the objects are that they will be working with, especially if they plan on using *BigBang* as a rubette in different contexts, as described under Point 6 above. In this case, they may decide to not select any objects, which means that the whatever they send the *BigBang* rubette, however many notes or sounds, the operations they defined will be applied to them. This way, they can experiment by applying the same

process to as many different inputs as they like.

This establishes an interesting connection to Lewin's theories. In his distinction between networks and graphs, the nodes of the former are associated with specific objects, while the nodes of the latter are not. *BigBang* precisely models the latter. We can design transformation graphs without having a specific application in mind, and then subsequently make them into networks when we send *BigBang* specific denotators.

7.1.2 The Temporal Existence of BigBangObjects

.

One of the major problems faced when implementing *BigBang* emerged from the problem of object selection. The idea of selecting something with a specific identity, and possibly even assigning it some characteristics such as visibility or audibility as described below, and moreover to expect that it remains selected even when transformed is hardly compatible with mathematical language. We have already discussed some aspects of this problem in the beginning of this thesis with respect to functions and the anti-Cartesian notion of transformation. What we saw there is that a function, mathematically speaking, does not really "move" its argument into the value, but rather associates a value, a new, different object, to the argument.

When dealing with computer software such as the *BigBang* rubette, we expect an entirely different behavior. When we select something, transform it, and continuously observe it during the transformation, we assume it to still be selected at any stage of the transformation, and thus the selected objects to maintain the same identity. However, since denotators are mathematical objects, this is not evident. *Rubato Composer*'s mathematical framework is implemented such that the transformed objects have a different identity from the original ones, which complies with the so-called

functional programming paradigm. In other words, denotators changed by morphisms or other operations such as insertion of factors, changing of values, and so on, usually yield new denotators rather than a modified original object, as would be expected in object-oriented programming. This is indeed justifiable in view of the workings of *Rubato Composer* and in view of denotators as mathematical structures. For instance, suppose a denotator was sent through several rubettes sequentially and the same denotator traveled through the whole system and was altered by each rubette. If the network was executed repeatedly, the denotator would be changed repeatedly, which does not comply with *Rubato Composer*'s principles. The functional paradigm is, not surprisingly, entirely incompatible with our notion transformations and gestures described in the first part of this thesis. In *BigBang*, transformed or manipulated objects should not yield new object identities, but change the ones we chose to transform. In other words, the value of an operation or transformation should be identified with its argument.

In order to get this functionality within *BigBang*, we need a representation of objects that observes what is going on mathematically and also keeps track of which object became which. This is another task that `BigBangObjects` can take care of. In the previous chapter we introduced `BigBangObjects` as simplifications of denotators that enable us to represent them visually. In addition to this, during the entire composition process, `BigBangObjects` keep track of where their corresponding denotators are. However, denotators are never saved within *BigBang*, but dynamically generated based on the input denotators and the operation graph as seen in the introduction to this chapter, and they frequently change. Thus, instead of remembering specific denotators or values, `BigBangObjects` remember a sequence of paths as their history – in the *Rubato Composer* framework, both forms and denotators are referred to by paths, as seen in Section 4.4.2 – for the topmost denotator identifying the object,

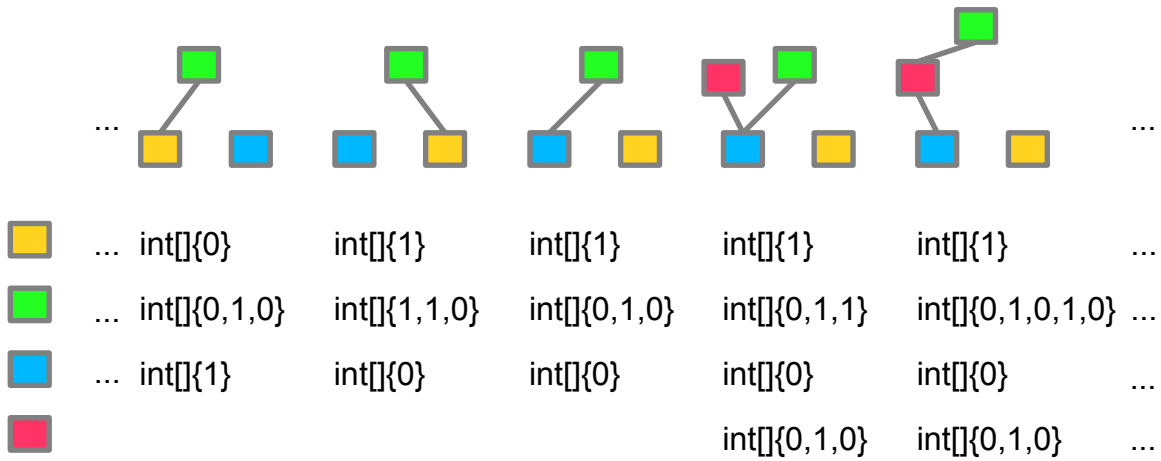


Figure 7.3: A table illustrating how BigBangObjects keep track of their location. Each column is a state of a simple composition process with an $FMSet$. The rows are what each of the objects save: a path for each of the state the object exists at, pointing to the denotators corresponding to the objects ($FMNodes$) are at, at the respective state. Note that all paths are assigned according to the x-axis here (*Loudness* in $FMSet$).

at any state of the composition. Since every `BigBangObject` is either the top-level object or an element of a **Power** or **List**, this path usually ends with an index in a **Power** or **List**.[3] Figure 7.3 shows how this works.

7.1.3 BigBangLayers

But `BigBangObjects` can do more. As seen with the architecture of sequencers or notation programs (Section 4.3.1), composers often think in tracks of musical objects, representing different voices or logical parts. One of the characteristics of *BigBang*'s facts view is that it does not distinguish such tracks in a spacial way. This leads to a significant gain in space. However, because of this, working in different parts can become tedious. However, with just the facts view, if users wanted to process several parts individually, they would have to repeatedly select each individual group

[3] In *Rubato Composer*, **Power** are sorted automatically and every of their elements can thus be referred to by a definite index.

of objects. In order to simplify this, we introduced so-called layers, which corresponds to tracks in sequencers, except for that they are not strictly tied to specific voices or instruments.

In early *BigBang*, layers were realized on the level of forms. All *SoundNotes* had an additional **Simple** denotator *Layer*, which represented the index of the layer on which the note was present. *SoundNotes* could be moved from layer to layer either by being transformed or with a specific menu function, and layers could be made invisible, and inactive, which rendered the notes unselectable.[4] This way of implementation had two disadvantages. First, the layers would only work for forms that contained the *Layer* **Simple** form. *MacroScores* and *Scores* could thus not be represented on layers. Second, the *Layer* form was of no use outside of *BigBang*, which hardly justified it to appear on the level of forms.

For the new generalized *BigBang* we had to find another solution. Layers were one of the main reasons for the introduction of `BigBangObjects` as temporal structures that were described in the previous section. We decided that each `BigBangObject` can be part of as many so-called `BigBangLayers` as needed. Users can add new layers, move objects to specific layers, or add them to additional ones. Each `BigBangLayer` can be made inactive (unselectable), invisible, and/or inaudible, which affects all of its objects. If an object is audible on at least one layer, it is audible. The same is true for visibility, and activeness. Figure 7.4 shows an example of an *FMSet* with active, inactive, inaudible, and selected layers.

[4]Thalmann and Mazzola, "Gestural Shaping and Transformation in a Universal Space of Structure and Sound".

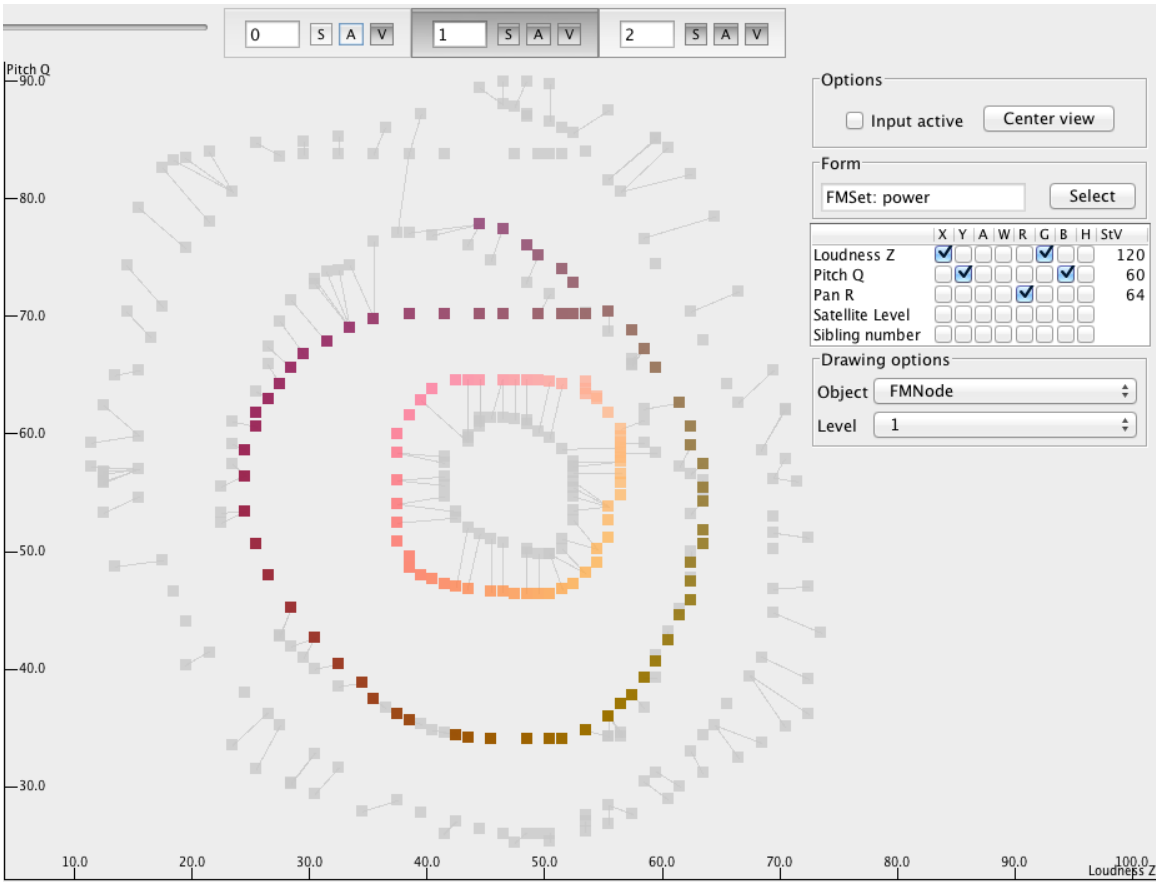


Figure 7.4: An *FMSet* distributed on three layers, represented by the rectangluar areas at the top. Layer 0 is inactive and inaudible (its *Partials* in the facts view are greyed out), layer 1 is active and selected (its *Partials* are darkened), and layer 2 is active, but not selected (normal bright color).

7.2 Operations and Transformations in BigBang

This section introduces all activities that are part of the graph represented in *BigBang*'s process view. In *BigBang*, we distinguish two types of processes, operations and transformations.[5] *Operations* are all activities that affect *BigBang*'s denotators, such as creating denotators, adding satellites, deleting denotators, etc. *Transfor-*

[5]Even though Lewin made the same distinction, our notion of operations greatly differs from what Lewin defined them to be. Rather than being more specific – bijective transformations – they are more general and denote the entire set of activities available in *BigBang* that change the contained denotators.

mations are special operations that include all activities that can be formulated as morphisms in the category of modules $\mathbf{Mod}^{@}$. Almost all operations and transformations are defined relatively to the x/y coordinates currently selected. This way, every dimension of the visible denotators can be manipulated. Since the way of interaction with *BigBang* is gestural for most of the operations, the next chapter, which deals with gestures, will explain how this works.

Almost all operations, except for the ones that add objects to the composition, are applied to a selection of `BigBangObjects` and keep references to these objects, rather than denotator paths, as they did in earlier *BigBang*. This has major advantages in case operations are modified, removed, or inserted, as will be discussed in Section 7.3.

7.2.1 Non-Transformational Operations

We do not consider all activities available in *BigBang* as part of the compositional or improvisation process. Only activities that change the denotator and the musical structure represented by the rubette are included in *BigBang*'s operation graph. For example, when the musician decides to select notes, move them from layer to layer, or pushes the play button, denotators are left unchanged. This section considers all operations that are not transformations.

AddObjects and DeleteObjects

The most basic operation is `AddObjects`, which is typically triggered when the user draws objects onto the screen or defines them using an interface, such as by recording with a MIDI keyboard or by defining points with the Leap Motion controller (see Section 4.3.3). All added objects are `BigBangObjects` and always comply with the form selected in *BigBang*. If the form defines several objects, on different satellite levels, users can choose which ones they wish to add. In *HarmonicSpectra* (Section 6.5.3),

for instance, they can choose between adding *HarmonicSpectrum* (elements of the top **Power**) or *Overtone* (elements of *Overtones*). Note that the latter cannot be added unless there is at least one *HarmonicSpectrum* already present.

Furthermore, objects can contain many **Colimit**s, as seen in Section 6.3.3. For each of those **Colimit**s, users need to choose which **Colimit** coordinate they would like to add. For instance in an *EulerScore* they can choose between adding *EulerNoteOrRests* with an *EulerNote*, or ones with a *Rest*.

For circular structures, uses can add objects to any satellite level, at most one level higher than the maximum level present. For instance, in case of a *SoundScore* with just one note, they can choose to either create *SoundNodes* (satellites) on levels 0 or 1, or *SoundNotes* (modulators) on level 1.

If there is no **Power** or **List** in the selected form, only one object can be added. Whenever users keep performing `AddObjects`, the former object is replaced. This happens for instance when the selected form is *Pitch* or *Note*.

Adding objects usually happens with respect to the selected x/y parameters. All denotator parameters not assigned to the x- or y-axes are given standard value, which can be defined by the user, in the column to the right of the view parameters checkbox grid. For instance, when drawing *Notes* on the $Onset \times Pitch$ plane, we can first decide that all *Voices* are 0, then continue drawing with voice 1, etc.

The `DeleteObjects` operation simply removes all objects currently selected.

InputComposition

Instead of adding denotators in any of the above ways, users can also input denotators by means of the rubette's input, which creates an `InputComposition` operation. For this, they need to connect the rubette to another rubette that outputs a denotator, for instance *MidiFileIn*, and run the *Rubato Composer* network. If there is already

a denotator in *BigBang*, the incoming one is of the same form, and the top-level denotator is a **Power** or a **List**, the elements of the incoming denotator are added to the denotator already present. Otherwise, the entire denotator is replaced. For instance, users can add as many other *Scores* to a *Score* as they wish, which leads to a large union, exactly the same way as can be done with the *Set* rubette. If they do not wish to do that, but replace the *Score* already present, they have to select and modify the `InputComposition` operation, as will be described in Section 7.3.2.

BuildSatellites and Flatten

When users want to build hierarchical structures, they can simply add objects by drawing on a higher satellite level, as seen above. However, they can also use the `BuildSatellites` operation, usually triggered through a pop-up menu, with which they can add objects that are already present in the composition as satellites to other objects, if the form contains **Power**s with the same coordinate form, at several places. This way, for example, with an $FMSet$ denotator with several top-level $Partials$ users can add some of these $Partials$ as modulators to another one of the $Partials$.

The opposite operation is called `Flatten`.[6] It adds all selected satellites to the **Power** or **List** that contain their respective anchors, e.g. it changes first-level $FMSet$ modulators into simple additive oscillators.

Shaping

`Shaping` allows users to change the values of the visible objects. It is based on two given denotator values u, v, for instance $Onset, Pitch$ and a number of real number pairs $(u_1, v_1), \ldots, (u_n, v_n) \in \mathbb{R}^2$. For each `BigBangObject` in the composition, if its u value is close to any u_n, its v value is assigned v_n. In practice, this is used in

[6] Guerino Mazzola et al. "Functors for Music: The Rubato Composer System". In: *Digital Art Weeks Proceedings*. Zürich: ETH, 2006.

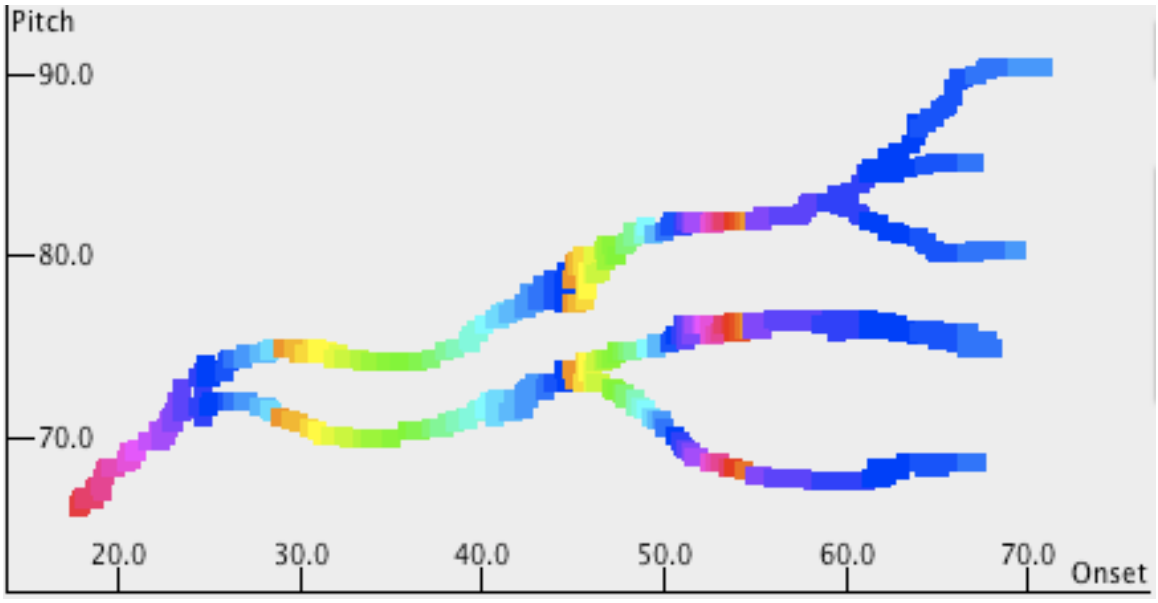


Figure 7.5: A composition drawn in *Onset* × *Pitch* with a shaped third dimension represented by color.

BigBang's shaping mode, where users can click-and-drag as in drawing mode on the x/y plane, and every object that is close to the x value is assigned the y value of the current drawing location.[7] This can be helpful especially with mouse drawing, where users are limited to drawing in two dimensions. After drawing on one plane (`AddObjects`), they can switch into shaping mode and define more dimensions the same way. Figure 7.5 shows a *Score* composition that was drawn on the *Onset*×*Pitch* plane, before being shaped in the *Onset* × *Loudness* plane, where for each *Onset*, a new *Loudness* was assigned. These varying *Loudnesses* are now represented with hue color values, red being both the loudest and quietest, green being *mp* and blue *mf*.

[7] An early version of this mode was introduced in Thalmann and Mazzola, "Gestural Shaping and Transformation in a Universal Space of Structure and Sound".

Wallpaper Operations

Wallpapers were introduced as generalizations of *Presto Ornaments* as mentioned in Section 4.4.1 and shown in Figure 4.10.[8] In simplified terms, for a wallpaper we need a *motif* m of coordinates of **Power**s or **List**s, a *grid* of morphisms $f_1, \ldots, f_n$ and corresponding *ranges* $r_1, \ldots, r_n$ with $r_k = (r_k^{min}, r_k^{max}) \in \mathbb{Z}^2$ and $r_k^{min} \leq r_k^{max}$. The first wallpaper dimension then results from the repeated application of $f_1(\ldots f_1(m))$ and creating the union of all copies. $1 + r_1^{max} - r_1^{min}$ determines the amount of copies of m we get. If $r_1^{min} \leq 0 \leq r_1^{max}$, m itself is included. The next dimension, if there is one, is then produced by f_2, r_2, applying f_2 to all copies of m resulting from the first dimension. Then f_3, r_3 to all results of f_2, r_2 and so on.[9] Figure 7.6 shows an instance of a two-dimensional wallpaper in early *BigBang*.

In *BigBang*, the morphisms of a wallpaper are limited to its transformations, i.e. affine morphisms, to be defined in Section 7.2.2. However, *BigBang* adds a functionality that the *Wallpaper* rubette was not capable of. The motif is no longer the entire composition, as for the *Wallpaper* rubette, but the selection the user has made. This way, elements on any level of the denotator anatomy can simultaneously participate in a wallpaper. For instance, we can generate a regular structure of $FMSet$ denotators including both, carriers and modulators. Each mapped modulator is added to its original carrier.

Also, each dimension of the wallpaper can be composed of several two-dimensional *BigBang* transformations, each of them performed on arbitrary x/y planes. This way, when working with a $Score$, a single dimension can for instance consist in a translation on the $Onset \times Pitch$ plane, followed by a shearing on the $Onset \times Loudness$ plane, which results in each copy of the motif being transposed in pitch, time, *and* loudness.

[8] Thalmann, "Musical composition with Grid Diagrams of Transformations".
[9] For more details, see ibid., p. 33f.

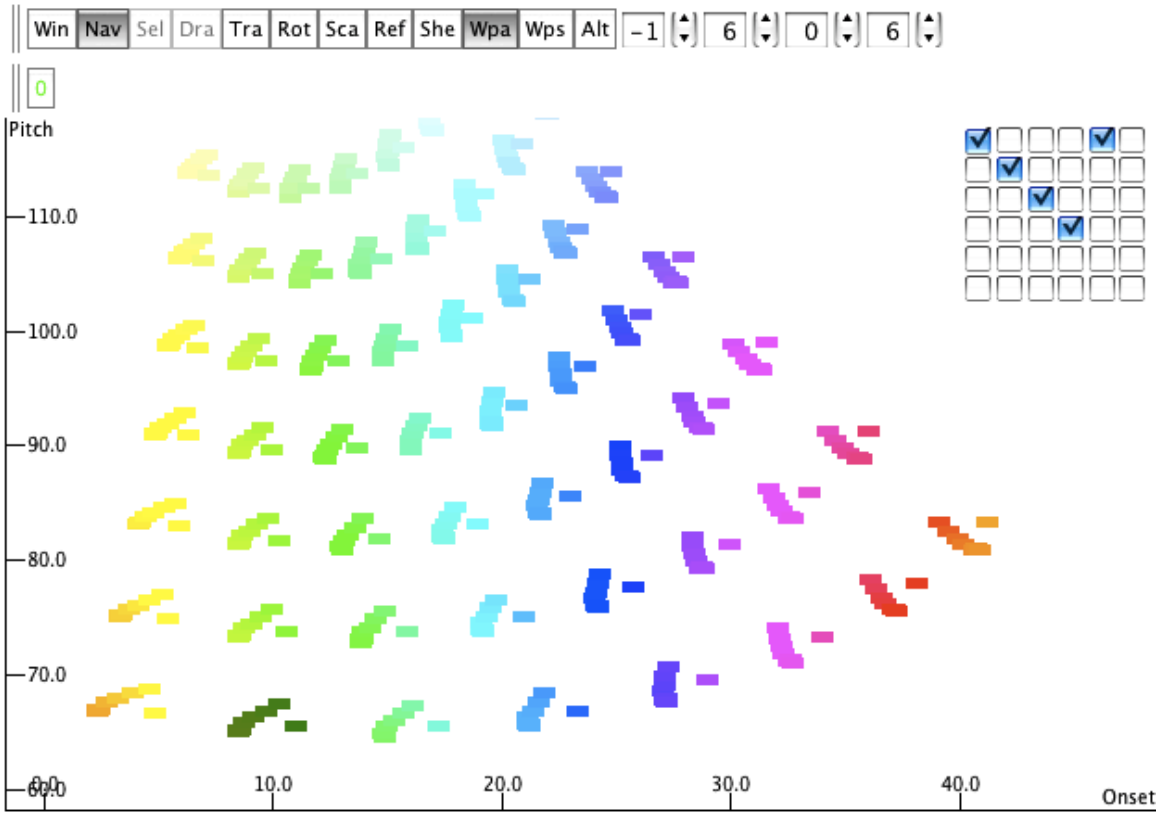


Figure 7.6: A two-dimensional wallpaper in early *BigBang*.

Two operations regulate the creation of wallpapers in the new *BigBang*, `AddWallpaperDimension` and `EndWallpaper`. Whenever `AddWallpaperDimension` is performed for the first time, all objects that are selected at the time are taken to be the motif, and all following transformations constitute the first dimension of a wallpaper. Each additional performance of `AddWallpaperDimension` adds another dimension, again constituted by all following transformations, while being based on the same motif. Finally, `EndWallpaper` ends the wallpaper and goes back to normal transformation mode.

Alteration

The last operation available in the current version of *BigBang* is `Alteration` and corresponds to the functionality of the *Alteration* rubette, a generalization of part of *Presto OrnaMagic*.[10] In *BigBang*, alteration consists in deforming a set of objects O_1 gradually towards another set of objects O_2. In *BigBang*, as with wallpaper motifs, both of these compositions can be selected using the selection tool and do not have to include the entire denotator. Again, compared to the *Alteration* rubette, where both inputs have to be **Power** denotators the direct elements of which are altered, in *BigBang* the sets of objects can include objects on different anatomical levels of the composition. For every object in $o_i \in O_1$, alteration finds the nearest object in $o_j \in O_2$, based on spacial distance, and moves the values of o_i towards the ones of o_j by a given degree. These degrees work as follows: for a degree of 0% the object o_i stays unchanged whereas for 100%, o_i becomes o_j.

Alteration can be performed simultaneously for as many of the denotator parameters as the user would like. For instance, if we merely alter *Pitch* in a *Score*, we get the tonal alteration familiar from music theory. If we alter just *Onset*, we get a generalized version of quantizing, familiar from sequencer systems. However, if we alter every **Simple** denotator in *Score*, we get intermediary compositions between O_1 and O_2. In the new *BigBang*, users can define two alteration degrees dg_1, dg_2 that act according to the denotator parameter currently associated with the x-axis. For instance, if we alter a *Score* while looking at the $Onset \times Pitch$ plane, dg_1 defines the degree by which the object with the earliest *Onset* is altered, while dg_2 designates the degree for the object with the latest *Onset*. If we switch to $Pitch \times Onset$, dg_1 concerns the object with the lowest *Pitch* and dg_2 the one with the highest. All degrees for the intermediary objects are interpolated linearly. Figure 7.7 shows an

[10] Thalmann, "Musical composition with Grid Diagrams of Transformations", p. 36f.

example from early *BigBang* where a *Score* is altered with $dg_1 = 0\%, dg_2 = 100\%$.

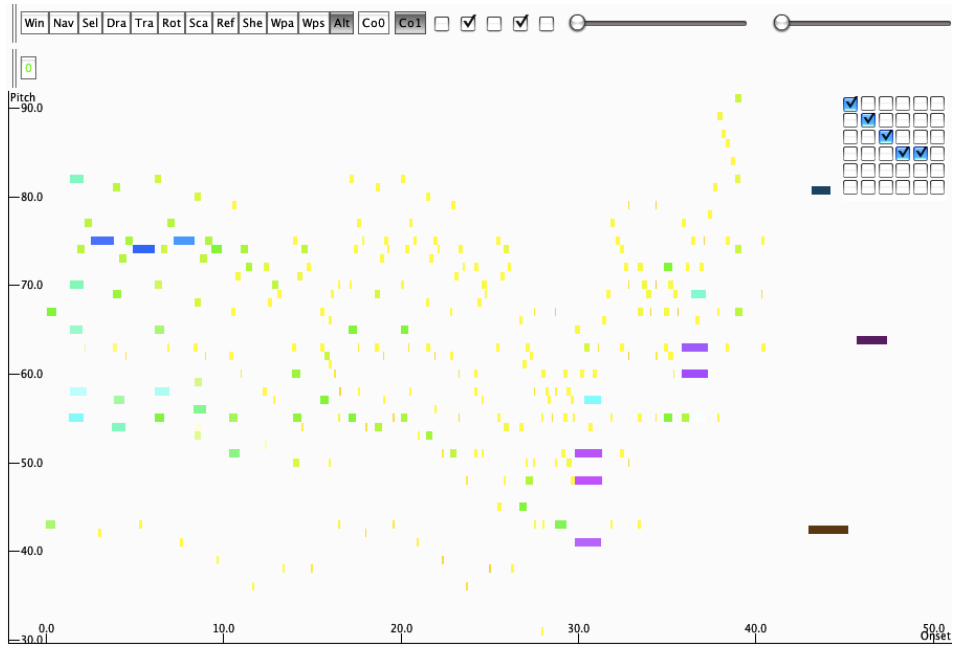
7.2.2 Transformations

In this thesis, transformations are directly based on morphisms between denotators, as we defined them above in Sections 3.2.2 and 3.2.3. *BigBang* allows for five different kinds of geometric transformations on the visible x/y plane: `Translation`, `Rotation`, `Scaling` (= dilation), `Shearing`, and `Reflection`, which takes advantage of a lemma that says that any multi-dimensional affine transformation can be described as a concatenation of such two-dimensional geometrical transformations. These transformations are typically applied with a gestural interface, as will be described in detail in the next chapter. For example, when using a multi-touch interface, users can directly define an `AffineTransformation` based on combined dilation, rotation, and translation with two fingers, or all five transformations with three fingers.[11]

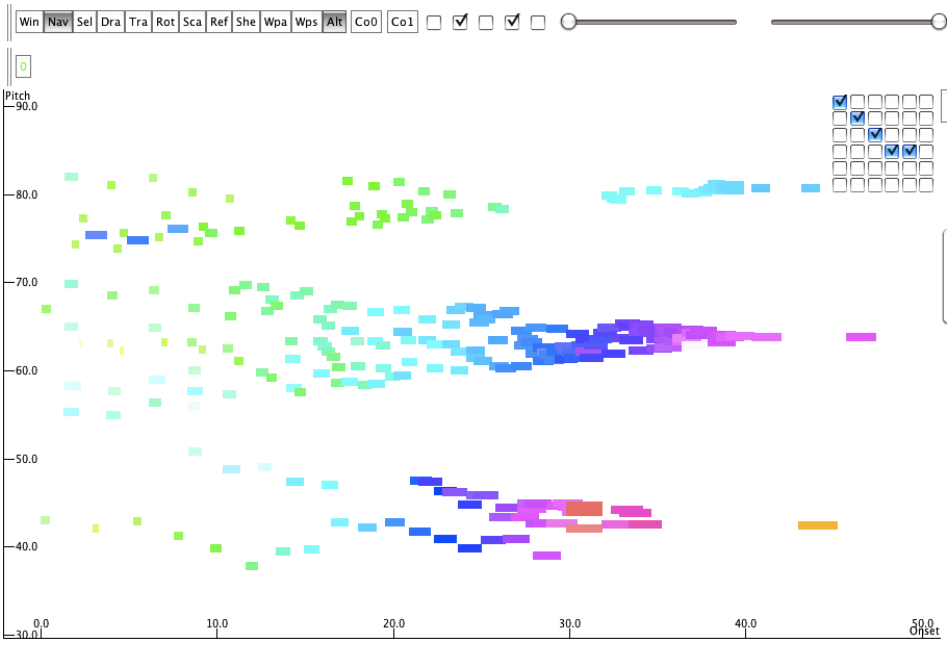
Regularly, transformations replace the selected objects with transformed versions. However, users also have a choice to perform so-called *copy-and-transform*, a generalized version of copy-and-paste, which adds the transformed objects to the given **Power** denotator, while keeping the originals. `Translation` with copy-and-transform yields classical copy-and-paste. Figure 7.8 shows a composition made with several copy-and-transforms.

In contrast to earlier versions, in the new *BigBang* rubette all transformations can be applied to any selection of `BigBangObjects`, on whichever anatomical level of the denotator they are, as seen above with wallpapers and alterations. This makes it possible for anchors and satellites to be transformed simultaneously, which leads to interesting results. Since satellites are designed to keep their relative position to their

[11] Florian Thalmann and Guerino Mazzola. "Affine Musical Transformations Using Multi-touch Gestures". In: *Ninad* 24 (2010), pp. 58–69.



(a)



(b)

Figure 7.7: A *Score* alteration in early *BigBang*. (a) shows the unaltered *Score*, whereas in (b) *Pitch* and *Duration* are altered with $dg_1 = 0\%$ and $dg_2 = 100\%$.

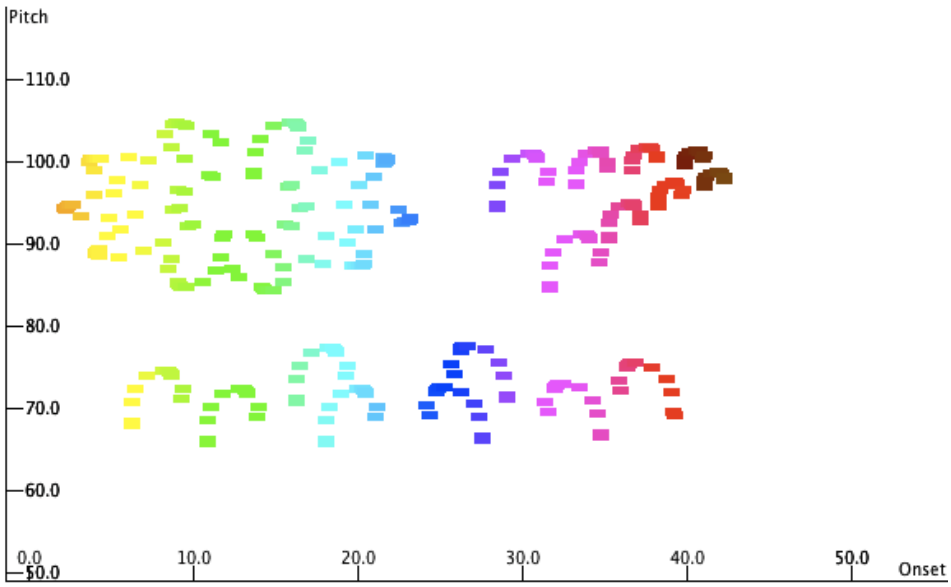


Figure 7.8: A small composition made with copy-and-translate (bottom left), copy-and-rotate (top left), copy-and-scale (top right), and copy-and-reflect (bottom right).

anchor when the anchor is transformed, a simultaneous transformation of anchors and satellites leads to satellites transforming doubly, once along with their anchor and once themselves.

Futhermore, even different objects in a **Colimit** can be transformed together in shared dimensions. For instance, in an *EulerScore* composition, we can transform *Notes* and *Rests* simultaneously if either *Onset* or *Duration* or both are associated with the x- and y-axes.

Since the view parameters can be freely assigned in *BigBang*, objects also need to be able to be transformed when only one of their parameters is associated with one of the visual axes. If this is the case, objects are represented on the respective axis and the transformation, defined in two dimensions, acts on the objects as though they were located in two-dimensional space. However, the results remain projections on the axis at any time.

Transformation in Arbitrary Spaces

Even though what the new *BigBang* rubette does in terms of operations and transformations may appear straightforward, from a theoretical point of view it is trickier than expected. In this section, I will briefly illuminate one of the solutions we found in order to deal with the potentially infinite number of object types that *BigBang* can handle.

Most importantly, the transformative system needed to be adjusted in order to transform more general types of **Simple** denotators, and not just *Note* parameters. For this, we could build on a procedure that allows for mapping denotators by arbitrary morphisms that I defined in my master's thesis and used in the context of the *Wallpaper*, *Alteration*, and *Morphing* rubettes.[12] Here I describe the necessary extensions and generalizations.

In the original procedure, the goal was to map a **Power** denotator d by a morphism f, even if the modules of its **Simple** denotators do not match the domain of f. This was done by inserting auxiliary *injection*, *projection*, and *casting* morphisms on both sides of f in order to adapt it to the chosen **Simple** morphisms.

We assume $f : V \to W$ to be any kind of affine or non-affine morphism where V and W are products of arbitrary modules $V = V_1 \times \ldots V_s$ and $W = W_1 \times \ldots W_t$. In the original procedure we tacitly assumed these modules to be one-dimensional free modules over the number rings $\mathbb{Z}, \mathbb{Q}, \mathbb{R}$ or $\mathbb{C}$. With the new extended repertoire of denotators, including **Limit**, **Colimit**, **Power** denotators based on **Simple** denotators on more-dimensional free modules as well as modules based on product rings, we needed to make some adjustments.

Assuming that we would like to map values of a given denotator $d : A@F$, where F is any form containing **Simple** forms, we again define two sequences $G. = (G_1, \ldots G_s)$

[12] Thalmann, "Musical composition with Grid Diagrams of Transformations", p. 32f.

and $H. = (H_1, \ldots H_t)$, their cardinality corresponding to domain dimension s and codomain dimension t of f. However, as opposed to the earlier procedure, their elements G_j and H_k are not **Simple** forms but either component modules of one- or more-dimensional free modules over a certain ring, or factors of direct sum modules or modules over a product ring. More formally, $G_j, H_k \in R_F$, where R_F is the set of all module components or factors throughout the denotator tree. There are significant differences between the set S_F introduced earlier,[13] and R_F. Not only does R_F contain modules and not simple forms, but it may contain several instances of the same type of component module or factor module, unless it is contained at the same position in a different instance of the same **Simple** form. There is thus no function analogue to $SA(S, d)$ involved.

Due to the fact that we now allow more-dimensional **Simple** denotators, we also need more auxiliary morphisms. In addition to i_j, p_k, g_j and h_k in the earlier procedure,[14] we define two additional sequences of projection and injection morphisms $\mathfrak{p}_m$ and $\mathfrak{i}_n$, which leaves us with the following collection of morphisms:

- the initial *projection* morphisms $\mathfrak{p}_1, \ldots \mathfrak{p}_s$ with $p_j : M_{G_j} \to G_j$,
- the initial *casting* morphisms $g_1, \ldots g_s$ with $g_j : G_j \to V_j$,
- the initial *injection* morphisms $i_1, \ldots i_s$ with $i_j : V_j \to V$ with $i_j(v) = v' = (0, \ldots v, \ldots 0)$, where v is at the j-th position of v',
- the final *projection* morphisms $p_1, \ldots p_t$ with $p_k : W \to W_k$ with $p_k(w) = w_k$ for $w = (w_1, \ldots w_t)$,
- the final *casting* morphisms $h_1, \ldots h_t$ with $h_k : W_k \to H_k$, and

[13] Thalmann, "Musical composition with Grid Diagrams of Transformations", p. 31.
[14] Ibid., p. 33.

- the final *injection* morphisms $\mathfrak{i}_1, \ldots \mathfrak{i}_t$ with $p_k : H_k \rightarrow M_{H_k}$,

In these definitions, M_{G_j}, M_{H_k} stand for the modules of which the G_j and H_k are components or factors. They do of course not have to be pairwise different, since several of the elements of $G., H.$ might be different components or factors of the same modules.

We then define a ϕ'_f analogous to ϕ_f:[15]

$$\phi'_f(d, (G_j)) = f(i_1 \circ g_1 \circ \mathfrak{p}_1 \circ A@M_{G_j}(d) + \ldots i_s \circ g_s \circ \mathfrak{p}_s \circ A@M_{G_j}(d)).$$

Finally, we define

$map_f(d)$ as a copy of d,
where every module M_{H_k} is replaced by the sum of $\mathfrak{i}_k \circ h_k \circ p_k \circ \phi'_f(d, (G_j))$,
and the injected projection of every component or factor M_i of M_{H_k}
with $M_i \neq H_k$.

7.3 BigBang's Process View

When they are performed, all of the operations and transformations described above are added to *BigBang*'s process view. In this section, we discuss how processes are visualized and how users can interact with them.

7.3.1 Visualization of Processes

As seen above, the process view shows a directed graph, which we call this graph *operation graph*, since it contains all operations performed, including transformations, and

[15] Thalmann, "Musical composition with Grid Diagrams of Transformations", p. 33.

since its node values are not defined in an absolute way and thus resemble Lewinian transformation graphs rather than networks (see Sections 3.2.1 and 7.1.1).

Whenever a new *BigBang* rubette is created or the user decides to start over by selecting a new form to work with, the operation graph is reset, which means that it merely consists of one node, labelled 0. For every operation performed, the graph obtains a new arrow, labelled with the operation, and a new node, representing the so-called `CompositionState` after the execution of the operation. Composition states are identified with unique increasing numbers, the highest of them representing the state last added. As long as the user merely interacts with the facts view, the graph grows as a linear sequence of arrows and nodes. Figure 7.9 shows such a simple linear graph including an `AddObjects` operation followed by all five geometric transformations.

7.3.2 Selecting States and Modifying Operations

Users can interact with the graph by selecting its nodes, which immediately updates *BigBang's* composition to the one at the corresponding state, both in the facts view, and the sonification. This way, the users can compare and contrast different states of their composition process, and evaluate them. Every time a state is selected, the shortest path between state 0 and the selected state is calculated and the corresponding facts are dynamically generated, which corresponds to the factualizing procedure described in Section 2.3.

When an operation is selected, the corresponding screen tool – a schematic representation of the operation as described in the next chapter – is shown, and users have the opportunity to modify the operation. Any state can be selected during this procedure and the consequences of the modification are shown for that state. This enables composers to change past decisions in their composition process, while observing

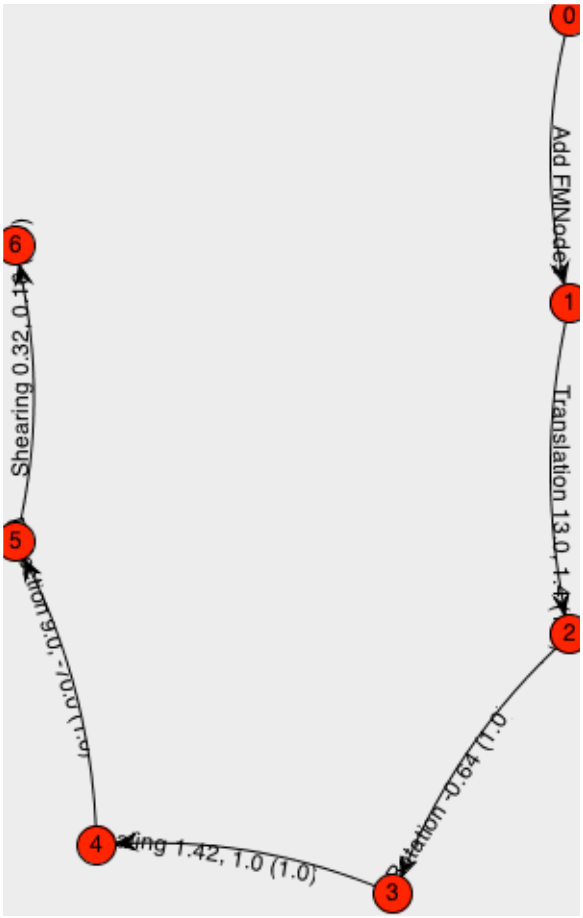


Figure 7.9: A *BigBang* operation graph showing a linear composition process.

their present composition, much in the fashion of Boulezian *analyse créatrice*, where composers use an analytical process to find other compositions in the neighborhood of theirs.[16]

Transformations are modified by dynamically changing the transformational parameters, e.g. the rotation angle or center for a `Rotation`, or the scale factors of a `Scaling`, which will be described in Section 8.1.4. Operations can have more distinct consequences. For instance, with modifying `AddObjects`, users can entirely replace the objects they were working with. The same composition process following the

[16]The notion of neighborhoods was introduced in Mazzola, *La vérité du beau dans la musique*, based on *analyse créatrice* in Pierre Boulez. *Jalons*. Paris: Bourgeois, 1989.

selected operation will then be applied to the new objects. The same applies to `InputComposition`. If the user selects such an operation before running the *Rubato Composer* network, the operation's composition is replaced. If no such operation is selected, a new `InputComposition` operation is created at the end of the graph or the selected composition state.

This is where the definition of operations with none of the objects selected becomes interesting, as described in Section 7.1.1. If a user replaces the entire input of *BigBang*, the entire composition process will be applied to all objects, no matter how many of them there are.

Now what happens when operations are modified that later operations depend on? If for instance, we modify a `Rotation` by a 180 degrees, all concerned objects' denotator paths may change, since they are based on lexicographical sorting, especially in **Power**. In early *BigBang*, this would have led to major problems, since all operations were directly based on denotator path references. In the current version, as mentioned in Section 7.2, operations keep references to `BigBangObjects` instead, which dramatically simplifies the case. In the case of the modified rotation, all paths the `BigBangObjects` refer to are changed. All operations following the rotation can then dynamically obtain the actual paths from the objects, when updating the denotator, i.e. during factualization.

7.3.3 Alternative and Parallel Processes

In addition to the linear processes described in Section 7.3.1, there are currently two more process types, alternative, and parallel processes.

If users select a state other than the latest composition state and perform an operation, the operation is added to the graph by building a fork at the selected state, building an *alternative process*. This way, users can experiment by building

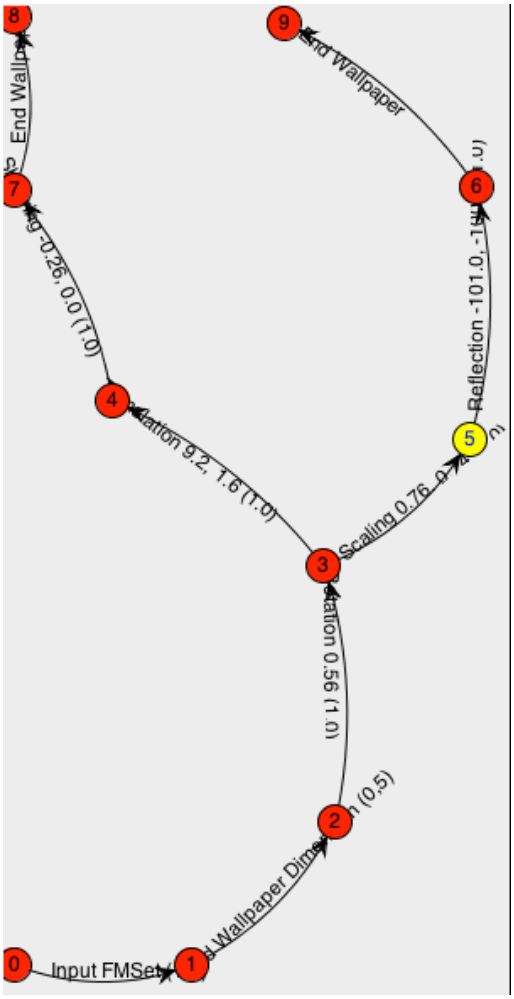


Figure 7.10: An operation graph with two alternative processes.

processes that share an initial part, but then continue individually. Such alternative composition states can again be selected and are immediately visualized and sonified accordingly. Figure 7.10 shows such a graph generating two alternative wallpapers starting from the same input material.

Parallel processes are created when an operation is selected in the graph. Then, any new operation performed is added as a parallel arrow to the selected operation, starting and ending at the same states. This is the only way operations are added to the graph without adding a new composition state. Logically, parallel operations are

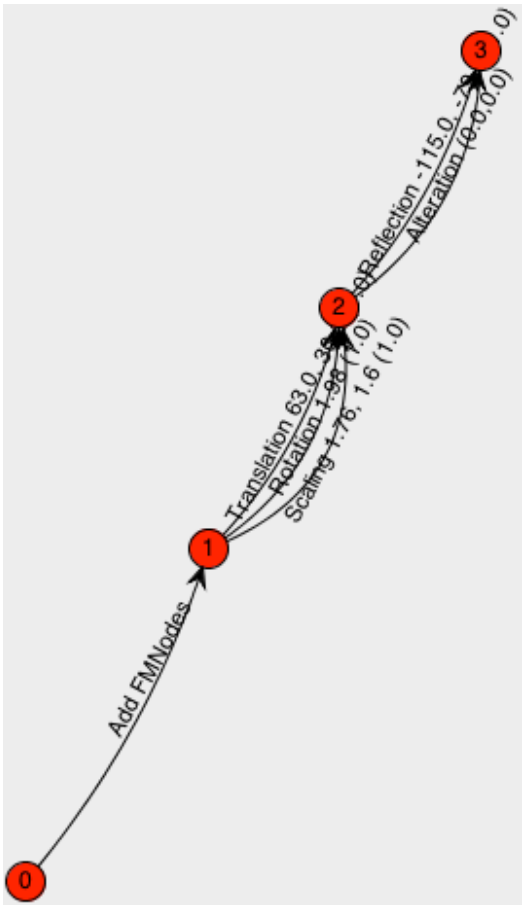


Figure 7.11: An operation graph with parallel processes.

no different from sequential operations at the current time. They are executed in their order of addition, since conflicting situations might arise with a parallel execution, especially with non-commutative transformations. However, as we will see in the next chapter, they differ from sequential operations in the way they are gesturalized (Section 8.2). Furthermore, they can be a good choice for composer to group their operations in order to get less composition states, if they are composing on a meta-level (see Section 10.2). Figure 7.11 shows a graph including three parallel geometric transformations, followed by a parallel reflection and alteration.

Currently, parallel operations can only be added as directly parallel to one operation. In the future, however, there will be the possibility to create higher-level

operations that are executed simultaneously to several lower-level operations. This is especially attractive for gesturalization, where higher-level operations would be animated much slower than lower-level operations. Internally, *BigBang* already supports the definition of such processes.

7.3.4 Structurally Modifying the Graph

So far I described ways in which operations can be added to the graph simply by performing them. There are additional ways in which users can interact with *BigBang's* operation graph.

Removing Operations

At any stage of the composition process, users can decide to remove an operation in the graph using a popup menu. When an operation is removed, all other operations are still executed and applied to the same selection of `BigBangObjects`. However, there is a chance that the concerned `BigBangObjects` are not there anymore, if the removed operation is for instance an `AddObjects` operation. This is why all operations are always applied to whichever of their objects are there, where all others are ignored.

Removing operations is an especially attractive solution to a problem that early *BigBang* had with its undo/redo system. Since its architecture was facts-based, as shown in the top half of Figure 7.1, non-invertible transformations such as projections were impossible to be undone. With the new, process-based *BigBang*, any type of operation can be undone without problems.

Inserting Operations

Users can also insert an operation at any state, by simply deciding where to insert and by selecting objects and executing an operation as usual. This replaces the selected

state node by two nodes, and connects them with an arrow representing the new operation.

Splitting Operations

Any operation can be split into two operations by indicating a ratio between 0 and 1 at which the operation should be split. They can do this using a slider, as described in the next chapter. Thereby, the operation arrow is replaced by two arrows and an intermediary node. For instance, if they split a `Rotation` with angle α at ratio 0.4, this results in two subsequent `Rotations` with the same center, the first with 0.4α and the second with 0.6α.

7.3.5 Undo/Redo

BigBang's operation graph already represents the composition or improvisation process. The possibility of interacting with it in the above ways may be seen as a replacement of traditional undo/redo functionality in software. However, on top of this, *BigBang* has a regular undo/redo system that works on the level of graph interaction. It allows users to undo and redo any activity of adding operations to and removing them from the graph. This is important for an even faster and more flexible way of interaction. For instance, if users decide to remove an operation early in the process and dislike the effect, they can bring it back using a standard key combination.

Chapter 8

Gestures: Gestural Interaction and Gesturalization

We have so far seen that the *BigBang* rubette allows users to visualize and sonify facts, and create and manipulate them using processes. In the last chapter, I also discussed that the only structures that *BigBang* represents internally are processes, only one of which refers to facts in the form of denotators (`InputComposition`). All other facts are generated dynamically, whenever an operation is added or modified. In order to offer an intuitive way of interacting with the software, we need yet another level: gestures.

BigBang builds on the gestural principles described in Section 4.3.3. There are two ways in which gestures come into play with *BigBang*, the ones that are performed by the user when applying operations, and the ones that are recreated from processes. With the former, anything composers and improvisers do within *BigBang* is immediately audible and most operations can be performed in continuous ways, using continuous physical controllers such as a mouse, a multi-touch surface, or a Leap Motion controller. All operations are accessible through a minimal amount of

actions or gestures, designed to be understandable to any user, even ones without a mathematical background.

Nevertheless, what *BigBang* saves are not the gestures as such, but their processual abstractions. From the point of view of computer science, this is an infinitely more economical solution than saving every temporal state of a gesture. The latter would be possible to be implemented with current computers, but it is not yet conceivable in terms of denotators and will thus be left to further projects.[1] Thus, the second way gestures are available in *BigBang* is by turning processes back into gestures, in the form of an animated composition history that can again be used for compositional purposes.

All this corresponds to the communication scheme between the levels of embodiment introduced in Section 2.3. The two types of gestures correspond to the inputs of the formalizing procedure and the outputs of the gesturalizing procedure. In this chapter, I explain how both formalizing and gesturalizing is implemented in *BigBang*.

First, an overview of gestural possibilities will be helpful. Table 8.1 lists all operations currently available in *BigBang* and shows whether or not their definition occurs in a gestural way (first type of gesture) and whether or not they are gesturalizable (second type of gesture). While several of the operations are not defined in a gestural way, most of them are gesturalizable. All transformations are both defined gesturally and gesturalizable. The two last columns will be discussed in Section 8.1.4.

[1] In order to do this properly within *Rubato Composer*, we have to extend its vocabulary to include constructs in the category of topological spaces, as used in the definition of gestures (see Section 3.3.2).

operation	defined gesturally	gesturalizable	modifiable	range (in $\mathbb{R}$)
`AddObjects`	yes	yes	yes	[0,1]
`DeleteObjects`	no	yes	yes	[0,1]
`InputComposition`	no	yes	yes	[0,1]
`BuildSatellites`	no	yes	yes	[0,1]
`Flatten`	no	yes	yes	[0,1]
`Shaping`	yes	yes	yes	[0,1]
`AddWallpaperDimension`	no	no	yes	[0,2]
`EndWallpaper`	no	no	no	no
`Alteration`	yes	yes	yes	[0,1]
`Translation`	yes	yes	yes	[0,2]
`Rotation`	yes	yes	yes	[0,2]
`Scaling`	yes	yes	yes	[0,2]
`Shearing`	yes	yes	yes	[0,2]
`Reflection`	yes	yes	yes	[0,2]
`AffineTransformation`	yes	yes	yes	[0,2]

Table 8.1: *BigBang*'s operations and their gestural capabilities.

8.1 Formalizing: From Gestures to Operations

In this section, I discuss the ways gestures are used to define operations, more precisely how controller gestures are mapped into appropriate operations and transformations. I thereby move from the most simple supported gestural interface to more complex ones. The standard gestural controller is the computer mouse. It was a design principle that almost everything in *BigBang* can be done in a satisfying way using a mouse. Other currently supported interfaces include multi-touch surfaces, the Leap Motion controller, and various MIDI controllers. As seen in Section 4.3.3, gestural devices vary significantly in the dimensionality of their topological space, the number of recognized parameters in this space, as well as the potential interdependency of the parameters based on physical limitations.

In our case, the mouse recognizes one point in $\mathbb{R}^2$, multi-touch a number of points in $\mathbb{R}^2$ (maximally 10 per user), and the Leap Motion twelve points and twelve vectors in $\mathbb{R}^3$. How do gestures look like in these spaces? For the mouse, for instance, we

can define a simple click-and-drag gesture $g :\uparrow\rightsquigarrow f$ where $\uparrow$ is the arrow digraph with $\uparrow = \bullet \to \bullet$ and $f : I \to \mathbb{R}^2$. Since we will always deal with single click-and-drag gestures below, we will simply identify the gestures by defining f.

8.1.1 Modes, Gestural Operations, and the Mouse

Complying with our principles for operation-based gestural systems, we decided the mouse operations in *BigBang* to be atomic gestures with as few clicks and movements as possible, so that they can be quickly applied, in an improvisational and potentially virtuosic way.[2] Most gestural operations can be defined with a click-and-drag gesture. In order to distinguish the different operations from each other we did not implement a recognition system, but defined a number of `Modes` in which the program can be, one for all gestural operations (see Table 8.1), plus one each for `AddWallpaperDimension` and `EndWallpaper`.[3] These modes are accessible through buttons in the top toolbar, but will soon be made accessible through keyboard shortcuts when using a mouse, or even a MIDI foot controller when working with two-handed gestural interfaces,[4] in order to keep the hands focused on gestures. Most modes for gestural operations have a corresponding `DisplayTool`, which represents the ongoing operation in a schematic way as a reference for the user. Figure 8.1 shows the tool displayed in `Shearing` mode, which consists of a grey square representing the original state and a clear parallelogram representing the sheared version of the square.

[2]Thalmann and Mazzola, "The BigBang Rubette: Gestural Music Composition with Rubato Composer", p. 3.

[3]There are also modes for non-operational activity, e.g. *Navigation* mode and *Selection* mode.

[4]As suggested in Daniel Tormoen, Florian Thalmann, and Guerino Mazzola. "The Composing Hand: Musical Creation with Leap Motion and the BigBang Rubette". In: *Proceedings of 14th International Conference on New Interfaces for Musical Expression (NIME)*. London, 2014.

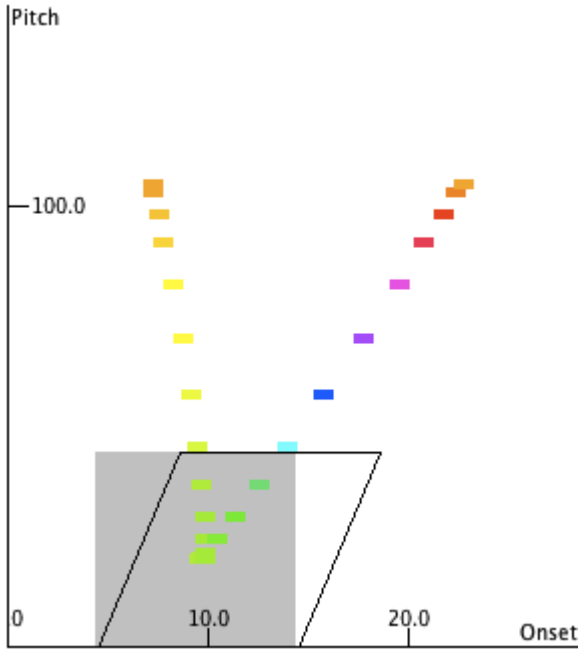


Figure 8.1: The shearing *DisplayTool* shown while a copy-and-shear is performed.

Gestural Transformations

The most interesting case of gestural control are transformations. In this section, I describe in a mathematical way how the user gestures are transformed into gestures on the canonical topological space of affine morphisms $\mathit{Aff}_2(\mathbb{R})$, and finally how we obtain the transformation morphism m that will be applied to the denotators represented by the selected objects.

All transformations in *BigBang* are currently two-dimensional affine transformations, which can be expressed as

$$y = Ax + b, \text{ with } A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \text{ and } b = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}.$$

For every mouse gesture, we find a gesture in $\mathit{Aff}_2(\mathbb{R})$ by a gesture morphism or throw morphism, which we defined earlier as a pair (u, v) with $u : \Gamma \to \Delta$ and

$v : X \to Y$ such that $h \circ u = \overrightarrow{v} \circ g$ (Sections 3.3.2 and 10.2). Since on both sides we deal with the graph with two edges and an arrow – this is what a simple click-and-drag gesture corresponds to – and thus $\Gamma = \Delta$, we can assume that $u(\gamma) = \gamma$ is the identity morphism on digraphs. All we thus need to do is define a $v : \mathbb{R}^2 \to \mathit{Aff}_2(\mathbb{R})$ for each transformation type.

In *BigBang*, each point of a click-and-drag mouse gesture is simply describable by two coefficients λ_x, λ_y, which represent motion along the x- and y-axes of the currently selected view configuration (see Section 6.2.1). Thus, λ_x, λ_y can stand for any of the denotator parameters. For simplicity, we assume here that the scale of the two axes directly correspond to the scale of the denotator parameters. In practice, however, depending on the currently selected zoom level, we need an additional conversion algorithm.

All transformations also depend on location, except for translation in our case. For this, users define an additional center point $c = (c_x, c_y)$. Unless indicated otherwise, the center is automatically defined by the initial click of the click-and-drag gesture. In order to execute an affine transformation relative to a center, we first need to translate by $-c$, then apply $Ax + b$ and finally translate back by c. This can be packed into a simple constant. If we assume $y = Ax + b$, then

$$y_c = y + (1 - A)c.$$

In the following discussion, we will omit this from the formulas for simplicity and only define $Ax + b$, even though c is always considered in *BigBang*. [5]

[5] For instance, for a scaling by λ_x, λ_y around c, see below, with

$$A = \begin{pmatrix} \lambda_x & 0 \\ 0 & \lambda_y \end{pmatrix} \text{ and } b = 0$$

Translation The most simple case is translation, where we can simply map the mouse space to the linear coefficient b. For this we define

$$v_T(\lambda_x, \lambda_y) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} x + \begin{pmatrix} \lambda_x \\ \lambda_y \end{pmatrix}.$$

Rotation Rotation is the only transformation that needs more than a click-and-drag gesture. First, users need to select a center around which to rotate, by simply clicking anywhere on the facts view. The center affects the rotation as seen above. Then a click-and-drag gesture decides over the rotation angle. Here, we need more than λ_x, λ_y. Two points (x_1, x_2) and (y_1, y_2) are the starting and current dragging or ending points of the click-and-drag gesture. We map as follows:

$$v_{Ro}(x_1, x_2, y_1, y_2) = \begin{pmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{pmatrix} x,$$

where ϕ is the angle around the center c determined by the angle between the straight from c to (x_1, x_2) and the one from c to (y_1, y_2).

Scaling For scaling, the initial click of the click-and-drag motion defines the center. λ_x, λ_y, defined by the dragging distance, determine the so-called *scale factors*:

$$v_{Sc}(\lambda_x, \lambda_y) = \begin{pmatrix} \lambda_x & 0 \\ 0 & \lambda_y \end{pmatrix} x.$$

---

we obtain

$$(1 - A)c = \begin{pmatrix} 1 - \lambda_x & 0 \\ 0 & 1 - \lambda_y \end{pmatrix} c = \begin{pmatrix} (1 - \lambda_x)c_x \\ (1 - \lambda_y)c_y \end{pmatrix}$$

and thus

$$v_{Sc} = \begin{pmatrix} \lambda_x & 0 \\ 0 & \lambda_y \end{pmatrix} + \begin{pmatrix} (1 - \lambda_x)c_x \\ (1 - \lambda_y)c_y \end{pmatrix}.$$

If the shift key is pressed, we set $\lambda_y = \lambda_x$ to allow for equal scaling in both dimensions.

Shearing Shearing works as the scaling does, where λ_x, λ_y define the *shearing factors*, where λ_x shears horizontally and λ_y vertically. We get the following formula

$$v_{Sh}(\lambda_x, \lambda_y) = \begin{pmatrix} 1 & \lambda_x \\ \lambda_y & 1 \end{pmatrix} x.$$

Purely horizontal or vertical shearing can be performed by pressing the shift key during the click-and-drag gesture. If $\lambda_x \geq \lambda_y$, we set $\lambda_y = 0$, else $\lambda_x = 0$.

Reflection Reflection is slightly more complex. The click-and-drag gesture determines the reflection axis rather than the positions of objects and reflected image, which could be done as well. We thus obtain

$$v_{Re}(\lambda_x, \lambda_y) = \begin{pmatrix} \frac{\lambda_x^2 - \lambda_y^2}{\lambda_x^2 + \lambda_y^2} & \frac{2\lambda_x\lambda_y}{\lambda_x^2 + \lambda_y^2} \\ \frac{2\lambda_x\lambda_y}{\lambda_x^2 + \lambda_y^2} & \frac{\lambda_y^2 - \lambda_x^2}{\lambda_x^2 + \lambda_y^2} \end{pmatrix} x.$$

Reflection is the only transformation that cannot be performed in a purely gestural way. We will see below that it can be easily gesturalized by interpolating through a projection on the axis. Here, however, we had to find a different solution. As soon as the initial click and a slight dragging motion is performed, the objects are abruptly reflected. However, as the user continues dragging, the axis is adjusted in a gestural way until a satisfying result is found.

Transformations in Wallpapers The initial and final operations that frame the execution of a wallpaper are not gestural. `AddWallpaperDimension` simply decides that all following transformations, until `EndWallpaper` occurs, will be part of the

wallpaper, and the two operations are executed by a simple click on the corresponding mode buttons. However, the way a wallpaper grows is always gestural, since the user applies regular transformations, executed as just described. For instance, if after an `AddWallpaperDimension` operation we start translating, we gesturally perform as many subsequent translations as the determined by the range of the wallpaper dimension. Every transformation we perform afterwards has a similarly gestural effect.

Other Gestural Operations

In addition to the transformations just described, there are also other operations that can be considered gestural.

Drawing For drawing with the mouse, which happens with the same click-and-drag gesture as above and triggers `AddObjects`, we can define a gesture morphism that, instead of going into the topology of affine transformations, directly reaches a topological space defined by the two denotator parameters associated with the x/y view parameters. For instance, if we draw $EulerNotes$ on the $Onset \times EulerPitch1$ plane (see Section 6.3.3), we can create a gesture morphism with u as above and $v : \mathbb{R}^2 \to \mathbb{Q} \times \mathbb{Z}$, if $Onset$ is defined over $\mathbb{Q}$ and $EulerPitch$ over $\mathbb{Z}$.

For each λ_x, λ_y, if we assume again a correspondence of view and denotator parameters as in Section 8.1.1 drawing defines an object with x/y parameters λ_x, λ_y. In reality, even though such a gesture morphism defines an infinite amount of objects, only a finite number are created due the discrete nature of mouse movements (pixel by pixel) in combination with a purposeful time constraint that limits the amount of objects drawn each second. However, by zooming in the facts view, objects can be created as closely together as necessary.

In sum, drawing could be considered the most gestural of all operations, since, for objects in **Power**s or **List**s, *BigBang* does not only remember the last state, as it is true for transformations, but create and remember all objects in order reached along the path. We will see later on that this has implications for gesturalizing, since we do not have to reconstruct a gesture but can in fact use this trace for gesturalizing.

Shaping Shaping works in a similar way to drawing, in terms of how it can be defined gesturally. The image space of the topological part v of the gesture morphism is also two-dimensional. However, while the second dimension is also the denotator space associated with the y-axis parameter, the first dimension is a discrete space defined by the set of all present values of the denotator parameter associated with the x-axis space.

Nevertheless, shaping remembers all shaping locations as elements of $\mathbb{R}^2$, which makes one shaping gesture applicable to any denotator, if for instance the input composition changes, or more objects are inserted at an earlier stage of the process.

Alteration When performing an alteration, users have gestural control over the alteration degrees dg_1, dg_2 (see Section 7.2.1) over two sliders in the top toolbar, which can be considered one-dimensional gestural controllers. Initially, both degrees are 0, which means that we see and hear the unchanged composition. Then, as soon as the sliders are moved, the composition O_1 moves continuously towards O_2.

The configuration with two sliders make it impossible with the mouse to control both degrees at the same time. However, this could be solved in the future using other controllers or a two-dimensional "slide field" instead of sliders, as it is used in many sequencing softwares.

Non-Gestural Operations

Several operations are not defined in a gestural way. Deleting, building satellites, and flattening, are all based on a selection of objects and happen at once, as described earlier on, upon a menu or keyboard command. For all of these, gestural versions are conceivable, but only partially implemented. For instance, deleting is possible in a gestural way when selecting an `AddObjects` operation and holding the shift key while clicking-and-dragging. This way, users can *undraw* notes previously drawn. In a similar way, instead of adding satellites, users can draw satellites simply by entering drawing mode and selecting the satellite level on which they would like to draw (described in Section 7.2.1. Despite their limited gesturality, these operations are all gesturalizable, as I will explain below.

The two framing wallpaper operations, as seen above, are the only operations that are neither gestural nor gesturalizable. They are simply discrete events with structural consequences for denotators and thus also need to be part of the process graph.

8.1.2 Affine Transformations and Multi-Touch

When using controllers other than the mouse, users also have the chance to directly define more general affine transformations. Such transformations combine all geometrical ones defined above. Before discussing how this works, I will briefly summarize how commonly used transformational multi-touch gestures work.

Multi-touch devices typically support the three gestural types *drag*, *pinch*, and *twist*, shown in Figure 8.2, which are all executed using two fingers and which correspond to the geometrical transformations translation, scaling, and rotation. Drag works the same way as the mouse gesture defined above, with the difference that

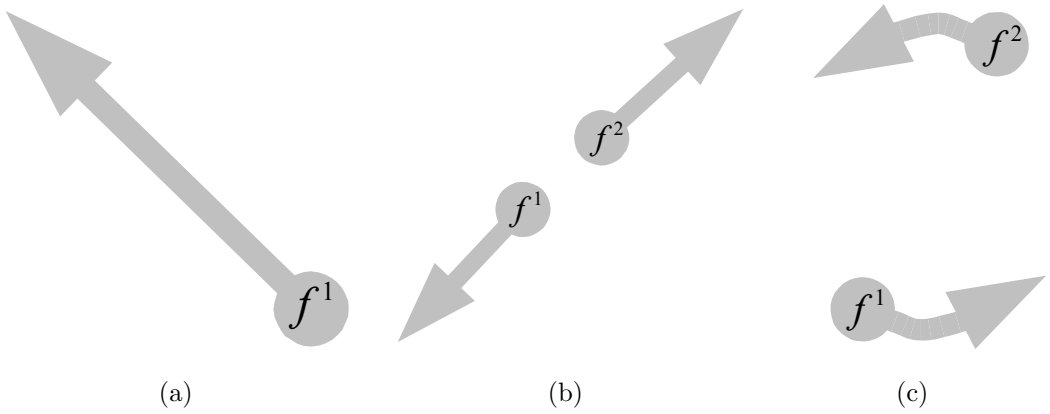


Figure 8.2: The three most common two-dimensional multi-touch gestures: (a) drag, (b) pinch, and (c) twist.

λ_x, λ_y are determined by the average position of the two fingers. In contrast, the gestural space of the other two gestures is not directly determined by finger position, but by a certain relationship between the two fingers used. For pinch, the distance between the two fingers determines a gesture on a one-dimensional topological space, and for twist it is the angle at which the fingers are placed that defines the topological space. Both gesture could thus be independently expressed as $g : I \to \mathbb{R}$.

Since these three parameters are all defined independently they can be used simultaneously. We can thus define a four-dimensional gesture $g' : I \to \mathbb{R}^4$ the components of which are $\lambda_x, \lambda_y, \lambda_p, \lambda_t$ for x-, y-position, pinch, and twist, with which we can simultaneously translate, scale, and rotate.

In an earlier paper, we generalized these gestures for two-dimensional affine transformations by adding a third finger.[6] We defined three fingers $f_i = (p_i^s, p_i^e)$ with finger indices $i = 1, 2, 3$ with starting point p_i^s and intermediary or ending point p_i^e, along with four vectors $v^j = p_2^j - p_1^j$, $w^j = p_3^j - p_1^j$ with $j = s, e$. Figure 8.3 visualizes these components. We also define the $d_i = p_i^e - p_i^s$ and $\hat{v}_j = \frac{v_j}{|v_j|}$. We then obtain the following gestural transformation parameters:

[6]Thalmann and Mazzola, "Affine Musical Transformations Using Multi-touch Gestures".

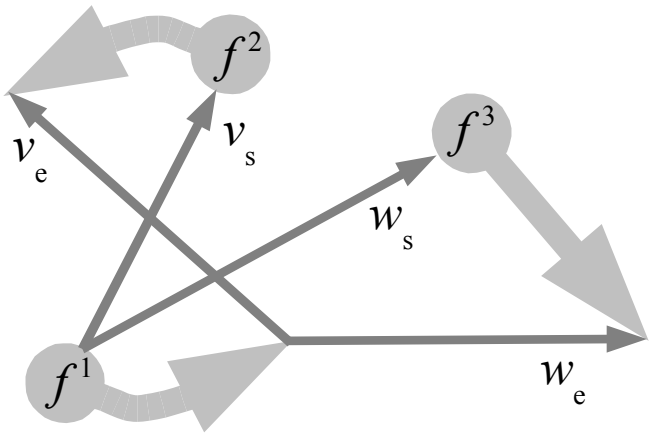
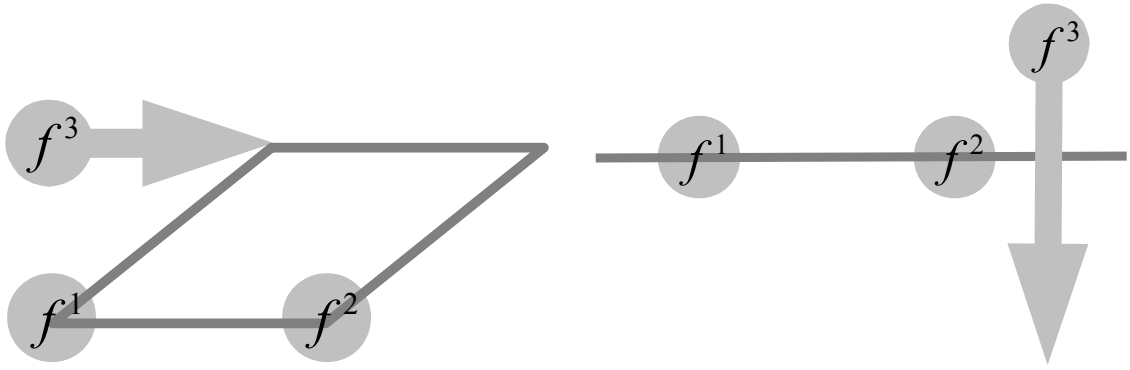


Figure 8.3: The components resulting from a three-finger gesture.

- As for two fingers, the *translation* component is defined by $\frac{d_1+d_2}{2}$,
- the *scaling* component is $\frac{v^e}{v^s}$, and
- the *rotation* component is $\arccos\left(\frac{v^e}{|v^e|} \cdot \frac{v^s}{|v^s|}\right)$.
- In addition to the above parameters, we obtain the *shearing* parameter, which is the projection length $|(d_3 \cdot \hat{v}^e)\hat{v}^e|$, and
- the *reflection* component defined by the projection length of $|(d_3 \cdot \hat{u}^e)\hat{u}^e|$, where $\hat{u}^e$ is a vector perpendicular to v^e.

With this, all geometrical transformations available in *BigBang* can be performed simultaneously. However, if we just wish to perform a reflection, we can hold fingers f_1 and f_2 steady at a distance, which can be seen as the reflection axis, and then move the third finger in a motion perpendicular to this axis (Figure 8.4 (a)). For a shearing, f_3 should move in parallel to the f_1-f_2-axis (Figure 8.4 (b)).

If we forget about the components just defined, we can move the three fingers around freely and perform any conceivable two-dimensional affine transformation, of course limited by physical constraints.



(a) (b)

Figure 8.4: The two three-finger gestures for (a) shearing and (b) reflection.

8.1.3 Dynamic Motives, Sound Synthesis, and Leap Motion

The most complex controller currently supported by *BigBang* is the Leap Motion controller (Figure 4.7 (b)). It can be used for precisely the same things that multi-touch can be used, including two-dimensional affine transformations.[7] *BigBang* thereby recognizes up to three fingers, projects them onto the plane perpendicular to the user's viewing direction (perpendicular to the z-axis at $z = 0$), and uses a procedure similar to the ones described in the previous section to perform two-dimensional affine transformations.

However, here we will be concerned with another functionality, in order to show the gestural possibilities of *BigBang*. Leap Motion can also be used to draw objects. In contrast to the procedure described above, where each gestural position generates an object, there is also the possibility to create and replace objects. When using the Leap Motion we treat each finger tip as a denotator and map the (x,y,z) location of each finger using a linear scaling into the coordinate system represented currently displayed by the *BigBang* rubette. Whenever the fingers move around the corre-

[7] Tormoen, Thalmann, and Mazzola, "The Composing Hand: Musical Creation with Leap Motion and the BigBang Rubette", p. 4.

sponding denotators are adjusted, which provides an immediate visual and auditive feedback. From there, we have the option to capture the currently defined denotators and keep adding new ones using the same method. If we use all three dimensions of the Leap Motion space, capturing is only possible with an external trigger (such as a MIDI trigger). To avoid the use of an external trigger the user can decide to use only two dimensions for drawing (preferably $x \times y$) and the third dimension for capturing, whenever a certain threshold, e.g. the plane perpendicular to the z-axis at $z = 0$, is crossed.

Figure 8.5 shows a situation where the modulators of a carrier in an $FMSet$ are defined using Leap Motion. Their arrangement directly corresponds to the fingertips in space, as can be verified visually. Compared to drawing with a mouse or another device, this method has significant advantages. The user can quickly compose complex musical structures while being able to smoothly preview each step until satisfied. Furthermore, the user can also easily edit musical objects added earlier in the process in the same continuous way which has many musical applications. The high precision of the Leap Motion makes this method just as accurate as using a mouse or trackpad.

One of the most useful musical applications of this way of generating objects, is to go back to editing the `AddObjects` operation, in the manner described in Section 7.3.2, after several transformations were performed. This way, users can gesturally redefine the motif that was transformed, and the entire following composition process is immediately applied to every gesturally changed state of the motif. This is especially interesting when the transformations consist in copying-and-transforming, which can yield an entire composition created from the same motif. Even more powerful is the use of the wallpapers to transform a motif, where the motif can virtually be grabbed by the user and moved around upon which the entire wallpaper moves accordingly.

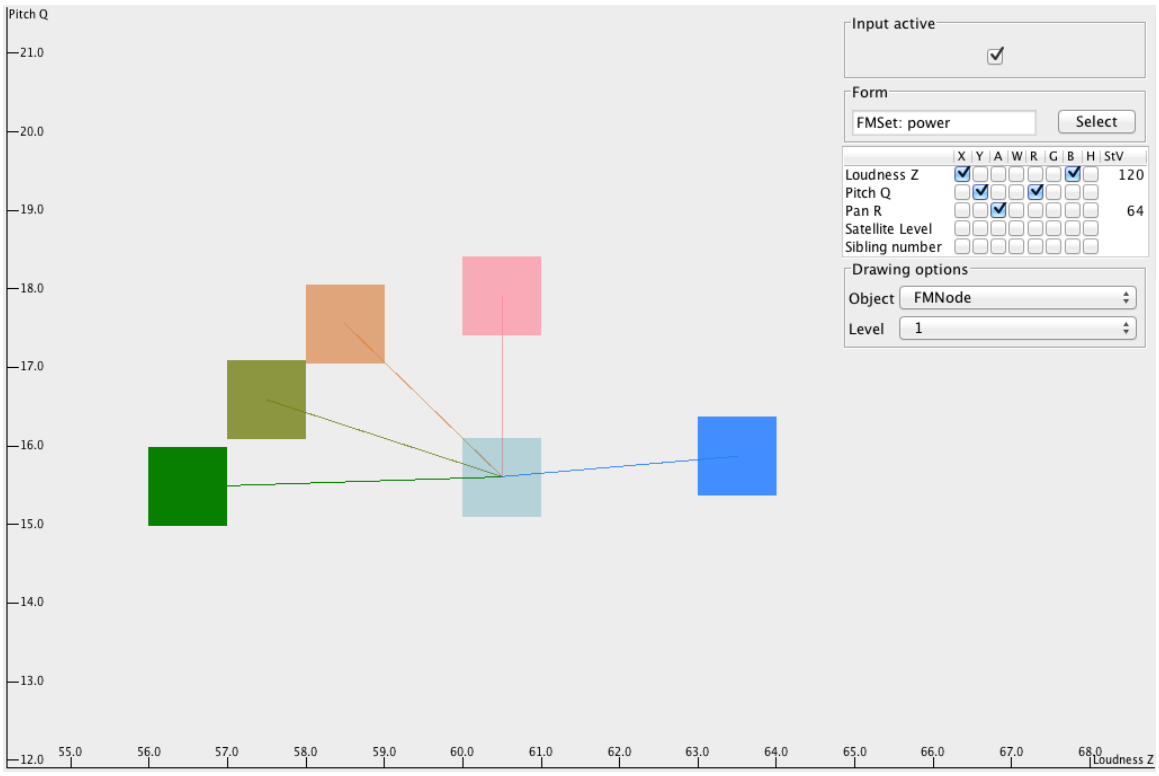


Figure 8.5: An *FMSet* denotator consisting of a carrier and five modulators defined by the fingertips of the user.

Figure 8.6 shows an example of such a wallpaper, where the motif has a recognizable hand shape defined by the user.

Instead of defining motifs in a composition or improvisation, users can also design sounds when choosing appropriate forms. For instance, while playing the keyboard, the positions of the fingers over the Leap Motion controller can be directly mapped to carrier oscillators or frequency modulators, as shown in Figure 8.5, and each hand movement changes their parameters. Furthermore, in a similar way, the user can create sounds and transform them gesturally in any of the geometrical transformation modes. This way, instead of changing simple parameters in a linear way as with commonly available synthesizer interfaces, multiple parameters can be changed in a complex way, such as for instance manipulating both frequency and amplitude of

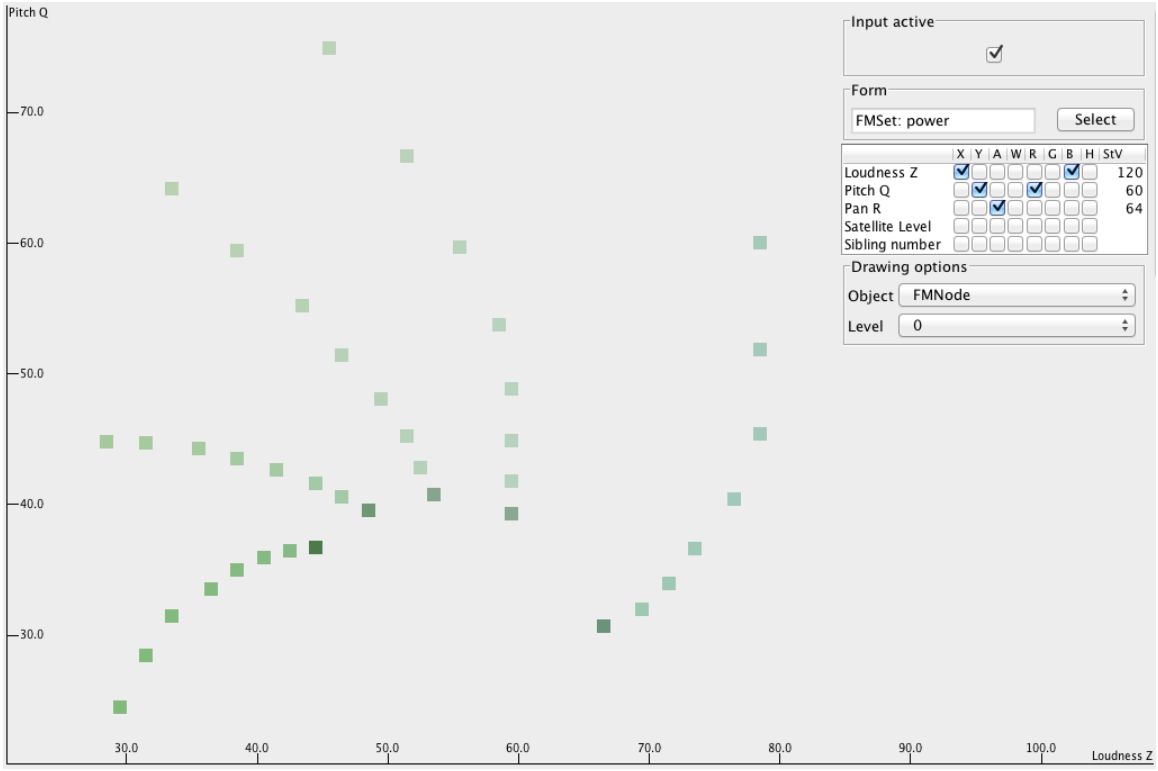


Figure 8.6: A wallpaper with a motif defined by the fingers of a hand.

hundreds of oscillators around a defined sound center.

This directly corresponds to the complex mapping strategies discussed in Section 4.3.3. Depending on form and transformation choice, users have almost infinite possibilities of dynamically mapping their gestural parameters to musical parameters. While translations map one-to-one, the other transformations have the potential to map a simple gesture to several parameters.

8.1.4 Recording, Modifying Operations and MIDI Controllers

Finally, here is a fourth way of controlling *BigBang* in a gestural way. Several types of MIDI controllers were made available, including keyboard controllers, mixing controllers, and combined ones. While keyboard controllers can be used to record MIDI notes into *BigBang*, not only by converting them into *Score* denotators but into any

denotators containing *Loudness* (from velocity), *Pitch*, or temporal **Simple** forms, in a similarly versatile way to the playback function discussed in Section 6.5.3. For instance, when working with a *Spectrum*, the temporal parameters of the MIDI input are ignored, while *Onset*/*Pitch* objects are added in a similar way to drawing mode.

While the above does not conform with the conditions for gestural control defined in Section 4.3.3 (note on/off events cannot be considered continuous), there are other uses of MIDI that are more gestural. The knobs and sliders on many devices send control changes that are gestural, even if in a discrete space ($g : I \to \mathbb{Z}$). Currently, such control changes are mapped to the modification of operations and transformations. All knobs and sliders are assigned to the operations in the order they were added to the graph. For instance, the 16 knobs of the *E-MU Xboard* are assigned to the 16 first operations, regardless of their occurrence in linear, alternative, or parallel processes. Since for each controller the control change assignments may vary, they all have to be configured individually.

For each control change, the sent values, integers within $[0, 127]$, are mapped real numbers within $[0, 2]$, where 0 corresponds to the identity, 1 to the original operation, and 2 to double the operation. The latter value is then used to replace the operation's values or morphism by a new one found at the corresponding point on the gesture. How this is done will be discussed in detail in the next section. For now, an example will suffice: if the modified operation is a rotation by 45°, MIDI value 31 will be mapped to 0.5, and will thus modify the angle to 22.5°, whereas $127 \to 2$ will lead to 90°. Almost all operations can be modified this way. However, only some, including all transformations, can be extended to double their amount. Table 8.1 shows which operations can be modified and lists all the ranges.

8.2 Gesturalizing and the Real BigBang: Animated Composition History

When we started designing the *BigBang* rubette, we chose *BigBang* as the working name for the prototype, because of the innumerable possibilities it brings to *Rubato Composer*. Meanwhile, this name has gained an initially unexpected and highly appropriate new meaning. The operation graph recounts the evolution of a sounding universe, which shows many parallels to the evolution of our physical universe. An initial group of musical objects expands and multiplies by being copied and transformed into a highly complex musical structure based on rules of symmetries. The ultimate functionality in *BigBang* is a gestural animation of this evolution, from the initial compositional actions to the actual state. In this section, I describe how this second type of gesture can be created.

As seen above, *BigBang* saves processes rather than gestures. From these processes, we can not only generate facts, but turn the processes back into gestures. The construct of a gesture ensures continuous and unidirectional motion in its topological space, by anchoring it in the interval I, as described in Section 3.3.2. Animating a gesture is thus straightforward: we just need to gradually interpolate on I, which gives us a sequence of points in the topological space, be it affine transformations ($\mathcal{Aff}_2(\mathbb{R})$) or any other structure. However, how do we get gestures from processes, which consist in merely a point in the corresponding topological space? I will start by answering this for transformations, and then move on to operations.

8.2.1 Gesturalizing Transformations

We saw that what *BigBang* keeps from the gestures performed by the user when transforming, are merely the ending points in the topological space, the final morphisms.

However, it also remembers which type of transformation the user was executing, which is helpful for reconstructing a standard gesture. As above, in the following definitions we ignore center c. In reality, we keep c constant during the entire gesture.

Translation

In Section 8.1.1, we saw that translations merely consist in the b-part of $Ax+b$. Thus, for a given translation

$$x + \begin{pmatrix} b_1 \\ b_2 \end{pmatrix},$$

all we need to do to create a gesture is define $g : I \to \mathit{Aff}_2(\mathbb{R})$ as follows:

$$g(i) = x + \begin{pmatrix} ib_1 \\ ib_2 \end{pmatrix},$$

for $i \in I$.

Rotation

For rotations, we interpolate on the angle ϕ and calculate the appropriate element of $\mathit{Aff}_2(\mathbb{R})$ as above. We thus define

$$g(i) = \begin{pmatrix} \cos i\phi & -\sin i\phi \\ \sin i\phi & \cos i\phi \end{pmatrix} x.$$

Scaling

For a scaling

$$\begin{pmatrix} a_{11} & 0 \\ 0 & a_{22} \end{pmatrix} x$$

we define

$$g(i) = \begin{pmatrix} 1 + i(a_{11} - 1) & 0 \\ 0 & 1 + i(a_{22} - 1) \end{pmatrix} x.$$

Shearing

For a shearing

$$\begin{pmatrix} 1 & a_{12} \\ a_{21} & 1 \end{pmatrix} x$$

accordingly

$$g(i) = \begin{pmatrix} 1 & ia_{12} \\ ia_{21} & 1 \end{pmatrix} x.$$

Reflection

Finally, we interpolate a reflection

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} x$$

by traveling through a projection on the reflection axis by doing the following:

$$g(i) = \begin{pmatrix} ia_{11} + (1 - i) & ia_{12} \\ ia_{21} & ia_{22} + (1 - i) \end{pmatrix} x.$$

Affine Transformations

The procedure for reflections also works for any affine transformation, since we interpolate between the identity matrix and any arbitrary matrix. What is missing in this formula is the b part, which we can deal with as discussed in the translation section.

Thus:

$$g(i) = \begin{pmatrix} ia_{11} + (1-i) & ia_{12} \\ ia_{21} & ia_{22} + (1-i) \end{pmatrix} x + \begin{pmatrix} ib_1 \\ ib_2 \end{pmatrix}.$$

However, the problem with this is that somewhere on the way, we might encounter singular projections that may not be musically optimal. In a former paper, we suggested the use of Bruhat standardized transformations in order to decompose affine transformations into their geometrical parts, and finally gesturalize on these obtained parts, which leads to more satisfying musical results.[8]

Gesturalizing Beyond the Transformation

As mentioned in Section 8.1.4, transformations cannot only be gesturalized in the interval $[0, 1]$ but even beyond it. The $i \in I$ in the formulas in this section can simply be replaced by an $r \in \mathbb{R}$, for which we get an extended gesture of infinite length. If we keep $r \in [0, 2]$, we get what we described above, and we can obtain exaggerated versions of the transformations, of up to double the amount.

8.2.2 Gesturalizing Other Operations

Almost all other operations can be gesturalized as well (see Table 8.1). We thereby distinguish so-called `ObjectBasedOperations` that operate on a single set of objects. They include `AddObjects`, `DeleteObjects`, `InputComposition`, `BuildSatellites`, `Flatten`, and `Shaping`. For these operations, we interpolate on the number of objects the operation manipulates. We define a function $o : I \to [0, n]$, n being the amount of objects, and $o(i) = in$. For instance, if an `BuildSatellites` operation adds 30 objects as satellites of any anchors, for $i = 0.2$ it only adds the first 6 objects. For operations that remember the order of their objects, such as `AddObjects`, the objects

[8] Mazzola and Thalmann, "Musical Composition and Gestural Diagrams".

are manipulated in order. This leads to an accurate reconstruction of a drawing gesture, as described above.

Two other operations can be gesturalized in a different way. Even though during gesturalization `AddWallpaperDimension` is ignored, since a wallpaper only evolves through its transformations. However, `AddWallpaperDimension` can be modified in the way described in Section 8.1.4. Then, $[0, 2]$ is simply used to adjust the upper range r^{max} of the wallpaper dimension, i.e. $o'(i) = i * r^{max}$ becomes the modified upper range. Finally, for `Alteration`, gesturalization affects the two alteration degrees dg_1 and dg_2. Thereby, $o''(i) = (idg_1, idg_2)$ is the modified pair of alteration degrees.

8.2.3 Using Gesturalization as a Compositional Tool

In *BigBang*, pressing on the *Gesturalize* button in the upper part of the process view, initiates a gesturalization of the shortest path that connects composition state 0 and the currently selected state in the operation graph, or the last added state, if no state is selected. Each gesturalizable operation along the way is gesturalized until the current state is reached. Users can specify an arbitrary duration for each operation. At each point in time, the current state is visualized and sonified as described above. Parallel operations are gesturalized simultaneously, despite their logical succession.

Even though gesturalizing can be used to reconstruct the composition process, it can become part of the composition itself. For instance, composers can design continuously evolving textures, by defining continuously sounding objects such as $FMSets$, transforming them in various ways, and finally creating a temporal structure by selecting various durations for the transformations. This way, the gesturalized structure becomes the actual composition.

This can also be done in a more improvisational way, by using a slider at the top of the process view. The space of the slider represents the entire gesturalization and

by moving the slider back and forth, users can continuously travel back and forth in the compositional evolution, while hearing the respective temporal states.

Part III

Implementation and Examples

Chapter 9

Architecture and Implementation

While in Parts I and II, I have discussed all conceptual matters and some decisions taken during the implementation of the *BigBang* rubette, I have done this from a more theoretical perspective, while discussing and showing examples from the rubette's graphical user interface. In this chapter, I will plunge deeper and describe *BigBang*'s architecture as well as some relevant implementation details. However, I will limit the discussion to the most significant aspects, since the code currently consists of more than 200 classes, more than 20000 lines of code, or almost 500 pages of font size 12, alone in *BigBang*'s package, along with many new and changed classes in other parts of *Rubato Composer*.

9.1 The Architecture of BigBang

As seen in Section 4.4, *Rubato Composer* offers both an extensive mathematical framework and an environment for quickly and simply creating additional rubettes. Since *BigBang* is more complex than any rubette implemented so far, it consists of an extension of the `AbstractRubette` class named `BigBangRubette`, which prescribes

less functionality than `SimpleAbstractRubette`, as discussed in Section 4.4.3. With `AbstractRubette`, the programmer has to entirely define the rubette's properties and view windows. Even though *BigBang* could be said to have more properties than any other rubette, its main purpose is to provide an interactive view of denotators, and the view is thus its central part. In order to simplify and unify the interface, we decided not to separate view and properties and provide access to all of the rubette's functionality uniquely through the view window. As described above, *BigBang*'s view is not limited to one view window. Once the main window is open, users may open several additional ones, each of them showing an different perspective on the composition.

These view windows determine the internal structure of *BigBang*. Due to the possibility of having multiple independent view windows, we decided to adopt a recursive or hierarchical *Model-View-Controller (MVC)* structure,[1] where the view is again structured as an MVC. This has the advantage that all views share the same main model, which contains everything that concerns the composition in *BigBang*: the denotators, `BigBangObjects`, operation graph, and so on. Each view then has the opportunity to have several components that access its own settings, such as `ViewParameters`, `DisplayModes`, synthesizer classes, and so on, through its controller. Figure 9.1 shows a simplified diagram of *BigBang*'s double MVC, where

[1] MVC is a popular combination of object-oriented design patterns, appropriate for programs with a graphical user interface. It neatly separates the main logic and data of a program, the model, from its potentially various graphical representations, the views. The latter communicate with the model through the controller, which offers access to a set of functions. In turn, whenever something in the model changes, it updates all its views to represent the current state. MVC was first published in Glenn E. Krasner and Stephen T. Pope. "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80". In: *Journal of Object-Oriented Programming* 1.3 (1988), pp. 26–49. Hierarchical MVC (HMVC) or Presentation-Abstraction-Control (PAC) has been suggested by several scholars, first by Joëlle Coutaz. "PAC, an implementation model for dialog design". In: *Proceedings of INTERACT*. 1987, pp. 431–6. However, it differs from the solution in this thesis by making all components uniquely communicate through their controllers. Here, the low-level view and the high-level model are identified.

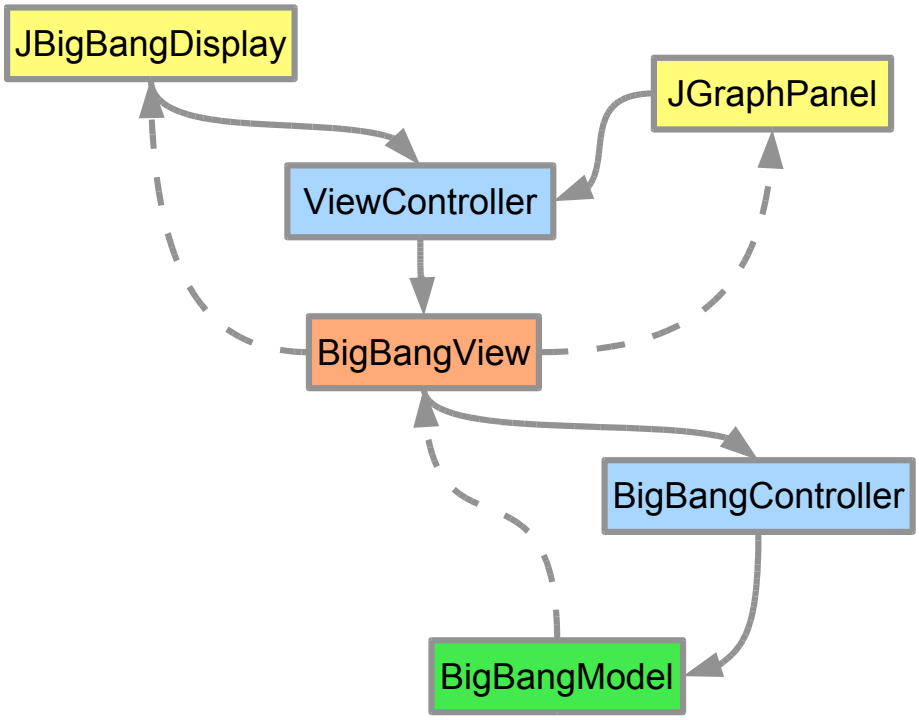


Figure 9.1: The general architecture of *BigBang*. Solid arrows refer to direct method calls, while dashed arrows show update procedures.

`BigBangView` is the main view, which in turn has two views, `JBigBangDisplay` and `JGraphPanel`. In reality, there are many more classes involved and in the following sections I will discuss the most important ones of them. I will start with the main model classes, followed by the view system, the main class of which is `BigBangView`. After that, I will discuss *BigBang*'s synthesizer and MIDI classes in a separate section, even though they are currently part of the view. Finally, I will discuss how we implemented some rubette-specific functionality, such as duplicating and saving, in *BigBang*. In our implementation, the controllers are kept as simply as possible – methods directly triggered by appropriate `Actions` and `ActionListeners` – and will thus not be discussed here.

BigBang's classes are organized into Java packages according to the MVC structure. The main package `org.rubato.rubettes.bigbang` contains subpackages `model`,

`view`, and `controller`, where `view` again contains `model`, `subview`, and `controller`. There are many more subpackages that will not be discussed here, such as `org.rubato.rubettes.bigbang.test`, which contains all classes with *JUnit* tests for most *BigBang* features.

9.2 BigBangModel

The *BigBang*'s model package contains the main logic of *BigBang* and manages the composition and all its structural forms of representation discussed in Part II. Since each *BigBang* holds exactly one composition – notabene in potentially many different states – there is only one instance of the model classes per instance of *BigBang*. `BigBangModel`, the main model class, uses and contains many other classes, the most important of which are shown in Figure 9.2. The responsibilities of `BigBangModel` can be divided into three main activities: creating operations and managing the operation graph (classes on the left in Figure 9.2), creating `BigBangObjects` and sending them to the views (classes in the middle), and updating and mapping the denotators as well as extracting their values (classes on the right).

From outside (through the `BigBangController`), the views have almost only access to methods that create and modify the graph, which is the main method of composing with *BigBang*. For instance, they can add a rotation of a certain number of selected objects to the end of the graph, or insert a satellite-building operation, or undo the deleting of an operation. Figure 9.3 visualizes in a simplified way what such an activity triggers. It depicts what happens when a translation is executed, which is initiated when the `BigBangController` calls the `translateObjects(...)` method of the `BigBangModel`. First, the model creates a new `TranslationTransformation` (step 1), which is a subclass of `AbstractOperation`, and adds it to a new

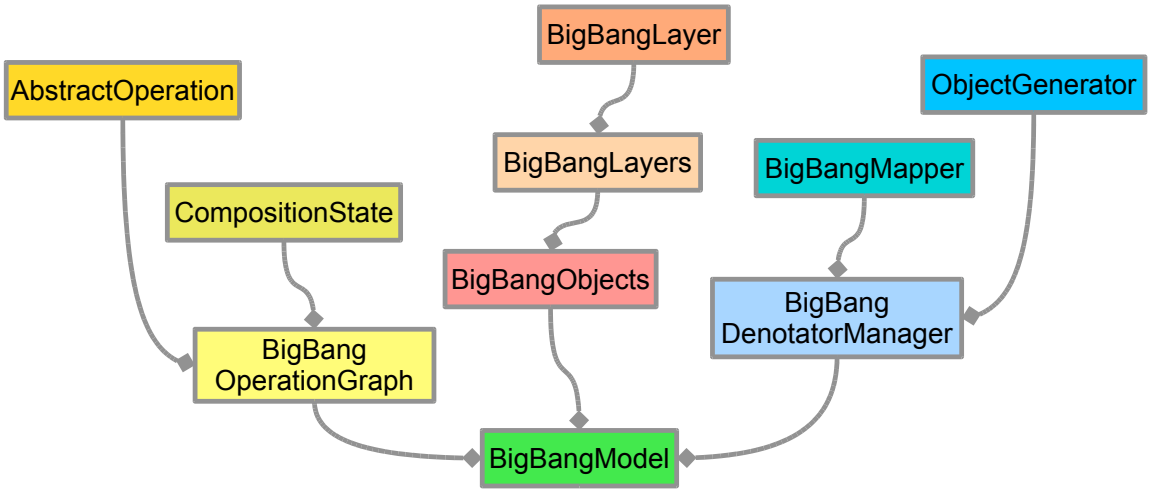


Figure 9.2: Some of the most important model classes of *BigBang*.

`AddOrInsertOperationEdit`, an `UndoableEdit` (step 2). The edit is then posted to *BigBang*'s undo-redo-mechanism (simplified here as `UndoSupport`, step 3), which allows any adding or removing of operations to be undone, and subsequently redone. When posted, the edit is automatically executed (step 4), which adds or inserts the translation to the `BigBangOperationGraph` (step 5), depending on which composition state is currently selected and depending on whether inserting mode is currently active (see Section 7.3.4). This adds a new edge to the graph and likely a new vertex, if the new edge is not a parallel edge.

Whenever a change is made to the graph or one of its operation, or when a different composition state is selected, *BigBang*'s composition is recreated accordingly, as discussed in the beginning of Chapter 7. Specifically, this is the task of the model's `updateComposition()`method, which first calculates the currently appropriate path in the graph, which contains the operations to be executed. For instance, if an intermediary composition state is selected, or one in an alternative part of the graph (see Section 7.3.3), the path is defined to be the shortest path ending at that state. Then, `BigBangModel` executes every operation of the given path in order (step 6 in

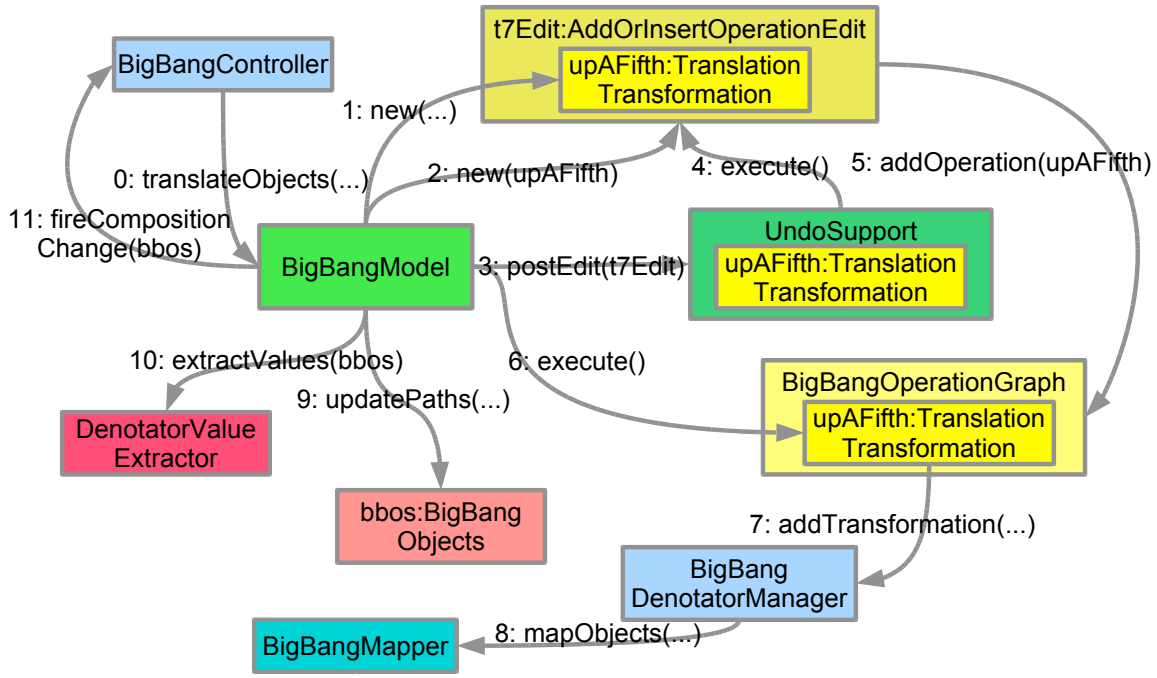


Figure 9.3: A simplified view of the process initiated by calling the *translateObjects(...)* method for a transposition T_7. The corresponding *TranslationTransformation*, named *upAFifth* here, is represented at different stages of its existence.

Figure 9.3). In our case, the `TranslationTransformation` is finally executed, which adds a transformation to the `BigBangDenotatorManager` (step 7). With each execution of an operation graph, the denotator manager resets its denotators (which comprise the actual composition), and then builds them up from scratch, according to the operations in the currently selected path. Simply put, the `addTransformation(...)` method asks the so-called `BigBangMapper`, the component that facilitates the mapping of arbitrary denotators in multiple dimensions (as described in Section 7.2.2), to map the denotators (step 8). After every execution of an operation, the denotator paths of the `BigBangObjects` are updated, depending of the changes made by the operation (step 9). For instance, a rotation of all objects by 180 degrees, reverses the order of all denotator paths.

When all operations of the current path are executed, the `BigBangModel` extracts

the values from the current denotators in `BigBangDenotatorManager` and updates them within its `BigBangObjects` (step 10). Finally, the model updates all views by sending them the current `BigBangObjects` (step 11), which they in turn visualize and sonify accordingly.

9.2.1 BigBangOperationGraph

BigBang's internal representation of the operation graph (`BigBangOperationGraph`) is implemented using the *Java Universal Network/Graph Framework (JUNG)*.[2] The framework provides functionality for modeling, analysis, and visualization of data in the form of graphs and networks. Most of what we described in Chapter 7 can easily be realized with *JUNG*. The framework does not only allow for interactive visualization with layout algorithms, navigation via zooming and panning, and graph editing, as used in *BigBang*'s process view (e.g. see Figures 7.9-7.11), but for instance also provides simple ways to obtain shortest paths between vertices, perform statistical analyses, or randomly generate or optimize graphs.

`BigBangOperationGraph` is realized as a `DirectedSparseMultigraph`, which means that it allows for parallel edges, as described in Section 7.3.3. *JUNG*'s graphs typically have typed vertices and edges, in our case `CompositionStates` and `AbstractOperations`, respectively. `CompositionStates` are currently kept maximally simple and merely keep track of their unique index number. `AbstractOperation`, on the other hand, is a superclass that represents any operation available in *BigBang*, as discussed in Section 7.2. Figure 9.4 shows the current class hierarchy. All `AbstractOperations` have attributes that determine whether they are gesturalizable or modifiable (Table 8.1), splittable (Section 7.3.4),

[2] Joshua O'Madadhain et al. *The JUNG (Java Universal Network/Graph) Framework*. URL: `http://jung.sourceforge.net/index.html`.

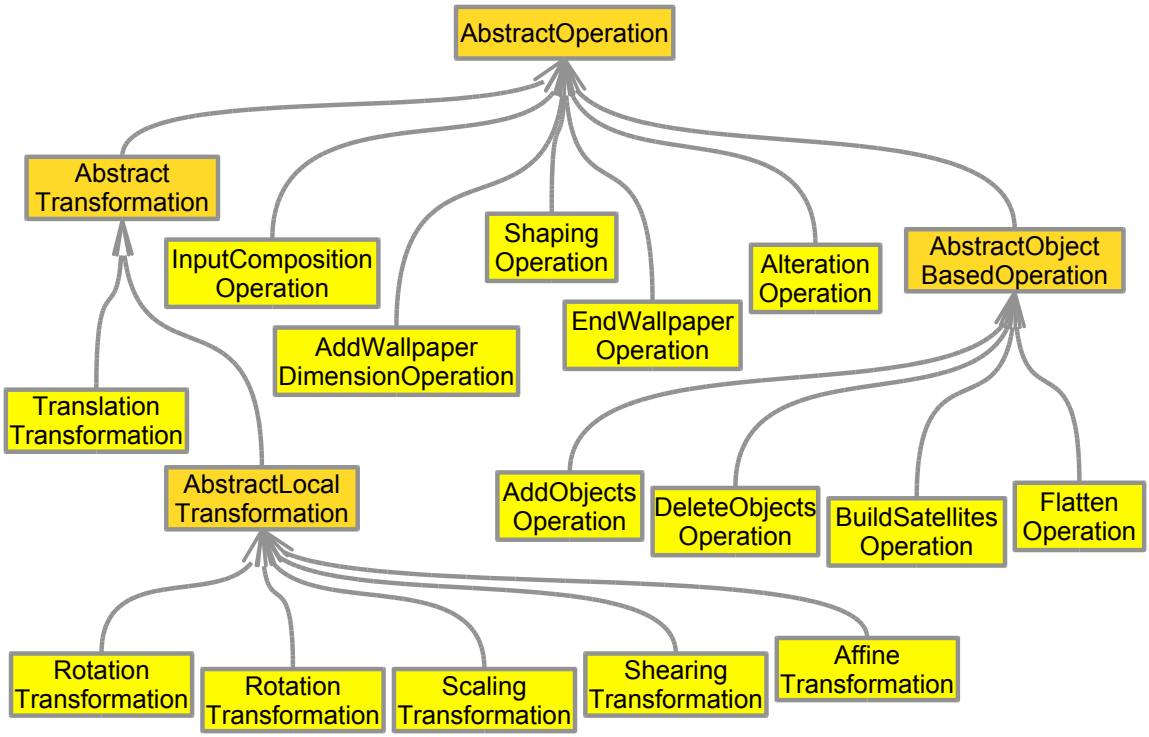


Figure 9.4: The class hierarchy of *BigBang*'s operations in *org.rubato.rubettes .bigbang.model.operations*.

and what their range of modification ratios, or their duration is during gesturalization. `AbstractTransformations` are all based on a number of `BigBangObjects` and a two-dimensional `ModuleMorphism`, while `AbstractLocalTransformations` adds the logic necessary for a shift to $(0, 0)$ and back that is necessary for most geometric and affine transformations, when performed relatively to a location different from $(0, 0)$, as discussed in Section 8.1.1. `AbstractObjectBasedOperations` include all operations that are based on a selected set of `BigBangObjects` and gesturalized by subsequently adding more and more objects to the set until it reaches its full size, as described in Section 8.2.2.

In addition to functionality provided by *JUNG*, the `BigBangOperationGraph` provides methods that allow for addition, insertion, and removal of `AbstractOperations`, while maintaining the necessary integrity. For instance, `CompositionStates` are al-

ways automatically inserted at the right position and they are always numbered in ascending order. If an operation is inserted at a position before the last composition state, it obtains an appropriate index, and the indices of all subsequent states are automatically increased. Also, `BigBangOperationGraph` keeps track of the order in which operations were added, or which operation or composition state is currently selected, which has implications on how what state of the composition is shown and where operations are inserted (Section 7.3.2).

As illustrated above, operations are added to and removed from the graph via corresponding undoable edits, `AddOrInsertOperationEdit` and `RemoveOperationEdit`. This system adds another level of compositional logic above the operation graph. It keeps track of all subsequently performed edits and allows users to navigate this history back and forth (Section 7.3.5).

The class responsible for the gesturalization of the graph is `BigBangGraphAnimator`. This class gets all operations on the shortest path to the currently shown composition state and gesturalizes each gesturalizable operation by modifying it with modification ratio 0 to 1 during the timespan given by the respective operation's duration, as described in Section 8.2. It also provides a method to jump to any position in the current gesturalization, which is for instance used by the gesturalization slider in *BigBang*'s process view.

9.2.2 BigBangDenotatorManager

In Chapter 7, we have seen that *BigBang* dynamically generates its composition as denotators, based on the information in the operation graph. Creating and managing these denotators is the task of the `BigBangDenotatorManager` and its helper classes in the package `org.rubato.rubettes.bigbang.model.denotators`. The denotator manager provides methods that correspond to all available ways of manipulating

denotators in *BigBang*. These methods are typically called by `AbstractOperations`, when executed by the `BigBangModel` (see Section 9.2). The following lists summarize the `BigBangDenotatorManager`'s most important public method declarations:

```
public OperationPathResults setOrAddComposition(Denotator composition);
public OperationPathResults addObjects(DenotatorPath powersetPath,
        List<Map<DenotatorPath,Double>> pathsWithValues);
public OperationPathResults removeObjects(
        List<DenotatorPath> removedObjectsPaths);

public OperationPathResults buildSatelliteObjects(
        Set<DenotatorPath> objectPaths,
        DenotatorPath parentPath, int powersetIndex);
public OperationPathResults flattenObjects(Set<DenotatorPath> objectPaths);

public OperationPathResults addTransformation(
        Set<DenotatorPath> objectPaths, DenotatorPath anchorPath,
        BigBangTransformation transformation);

public void addWallpaperDimension(Set<DenotatorPath> objectPaths,
        int rangeFrom, int rangeTo);
public void endWallpaper();

public OperationPathResults addAlteration(
        Set<DenotatorPath> foregroundComposition,
        Set<DenotatorPath> backgroundComposition,
        List<DenotatorPath> alterationCoordinates,
        double startDegree, double endDegree,
        DenotatorPath degreesDimensionPath);

public OperationPathResults shapeObjects(Set<DenotatorPath> objectPaths,
        TreeMap<Double,Double> shapingLocations,
        List<TransformationPaths> shapingPaths,
        boolean copyAndShape);
```

Most of these methods take `DenotatorPaths` as arguments, which are sophisticated representations of the `int[]` paths discussed in Section 7.1.2, and which correspond to the main paths of the `BigBangObjects` to be manipulated. As discussed above, the path at which each `BigBangObject` can be found varies from composi-

tion state to composition state and it is one of the tasks of the `BigBangObjects` to keep track of these paths. Whenever an `AbstractOperation` is executed, all `BigBangObjects` in question are asked for their current path and these paths are forwarded to the `BigBangDenotatorManager`. Almost every of the denotator manager's methods return an `OperationPathResults` object, which contains all paths added, changed, and removed by the operation. This, in turn, is then used to update the `BigBangObjects`' path tracking system, as will be discussed later on. The denotator manager contains additional public methods, some of which are:

```
public void setForm(Form baseForm);
public Form getForm();
public boolean isFormCompatibleWithCurrentForm(Form form);

public void reset();
public OperationPathResults getPathResults();

public BigBangDenotatorManager clone();
public Denotator getComposition();
```

The first three are concerned with the current form of *BigBang*. The form can be replaced, which leads to a reset of the denotators as well as the operation graph. The `reset()` method is also called before every execution of the graph and thus whenever an operation is added to the graph or modified (see Section 9.2). `getPathResults()` can be called to obtain temporary `OperationPathResults` after partial executions of operations. The last two methods are used by the model when the rubette is duplicated (see Section 9.5) or when the current composition is sent on to another rubette via *BigBang*'s output.

Other methods of the denotator manager are merely visible within its package `org.rubato.rubettes.bigbang.model.denotators` and are used by the helper classes. They allow for consistent adding or removing objects, moving them to satel-

lite sets, and so on. Most importantly, the `BigBangDenotatorManager` always ensures that objects are automatically made relative when they become satellites, and absolute when they become top-level objects, a functionality otherwise only provided by specific rubettes. Some of the methods available within the package are:

```
List<Denotator> addObjects(List<Denotator> objects,
        List<DenotatorPath> parentPaths, int[] powersetIndices);
List<Denotator> addObjectsToParent(List<Denotator> newObjects,
        DenotatorPath powersetPath);
void replaceObjects(List<Denotator> newObjects,
        List<DenotatorPath> replacedObjectsPaths);
void replaceSiblingObjects(List<Denotator> newObjects,
        List<DenotatorPath> replacedObjectsPaths);
List<Denotator> getAbsoluteObjects(List<DenotatorPath> objectPaths);
Denotator getAbsoluteObject(DenotatorPath objectPath);
```

The helper classes used by `BigBangDenotatorManager` are shown in Figure 9.5. The most important one is the `ObjectGenerator`, which offers simple ways of creating denotators, for instance using the methods:

```
public List<Denotator> createObjects(Form objectForm,
        List<Map<DenotatorPath,Double>> pathsWithValues);
public Denotator createStandardDenotator(Form form, double... values);
```

All other helper classes execute the mathematical operations and generate the structures in similarly simple and consistent ways, all offering methods that allow programmers to think on the level of objects such as in `BigBangObjects`, rather than denotators. For instance, `BigBangMapper` enables direct and simultaneous transformations of anchors and satellites, even though the latter are defined relatively to an anchor, whereas the former are absolute.

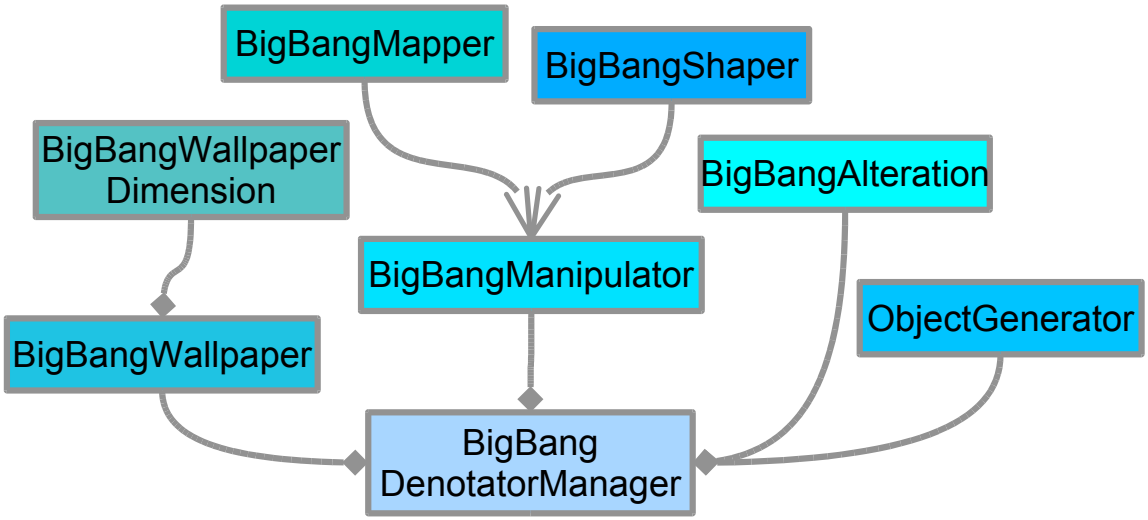


Figure 9.5: *BigBang*'s denotator manager and its helper classes in package *org.rubato.rubettes.bigbang.model.denotators*.

9.2.3 BigBangObjects

Finally, the third group of model classes deal with `BigBangObjects`, the simplified and historical representation of denotators discussed in Section 7.1.2. Its main class is called `BigBangObjects`, which contains maps that keep track of which `BigBangObject` instances are present before which `AbstractOperations`, as well as the `DenotatorPath` they are temporarily associated with, in order to allow for more optimized queries rather than with asking the respective `BigBangObject` instances. The class `BigBangObjects` also offers methods that return analytical data, used for instance for visualization, such as denotator value names, minimum and maximum values for each denotator value, maximum satellite level, and so on. The most important method is

```
updatePaths(AbstractOperation previousOperation,
        AbstractOperation operation, OperationPathResults pathResults),
```

which is called after every time an operation is executed and `OperationPathResults` are returned by its `execute()` method. It adjusts all paths of added, changed, and removed `BigBangObject` instances.

The `BigBangObject` class, in turn, keeps track of a specific object's `DenotatorPath`, parent, and children, at every `AbstractOperation` it occurs. Also, it holds the denotator values of the currently shown state (found by the `DenotatorValueExtractor`, see below), as well as the set of `BigBangLayers` the object appears on. Besides methods managing this data, `BigBangObject` also offers methods to query its activeness, audibility, and visibility (see Section 7.1.3), comparison methods, and methods for finding specific denotator values.

A crucial class for the `BigBangObject` is the `DenotatorValueExtractor`. After all operations in the currently active path of the graph are executed, the extractor is asked to gather all denotator values from the currently represented composition, which is obtained through the `getComposition()` method of the denotator manager. The extractor then updates the denotator values in each of corresponding `BigBangObject` instances. It contains no public methods and simply does its task upon construction using the main constructor:

```
DenotatorValueExtractor(BigBangObjects objects, Denotator composition);
```

9.3 BigBangView

The view classes of *BigBang* currently only contain one type of main view, `BigBangView`. It comes in two versions, one for the standard mouse-operated version, using Java Swing for visualization, and one for multi-touch interfaces, using

OpenGL through the *MT4j* framework.[3] The latter's main class is an extension of the regular `BigBangView`: `MTBigBangView`. As seen above, both of these versions also allow for additional interfaces, such as the Leap Motion, or MIDI controllers. Here, we focus on the mouse-operated version.[4]

We have already seen that `BigBangView` is in fact again a model, associated with several (sub)views. Figure 9.6 compiles the most important classes involved in this system. All classes represented in the upper half of the figure communicate any user activity through actions and adapters to the `ViewController`, which updates the model classes represented in the lower half of the figure. Whenever a model class is changed, it updates all its observers, again through `ViewController`. For instance, `JBigBangDisplay` visualizes `DisplayObjects` and gets updates whenever these change. `DisplayObjects`, in turn, are visual analogues of `BigBangObjects` (see below) and are updated through `BigBangController` whenever *BigBang*'s composition changes (see Section 9.2). Other views might also take information from the `DisplayObjects` and observe them in a similar fashion.

9.3.1 The View's Model Classes

Figure 9.6 shows several of *BigBang*'s view model classes located in the Java package `org.rubato.rubettes.bigbang.view.model`. The most important ones are the `DisplayObjects` and `DisplayObject`, which are visual representations of `BigBangObjects` and `BigBangObject` and directly refer to those, especially their current denotator values gathered by the `DenotatorValueExtractor`, as shown in Section 9.2.3. We first saw in Section 6.2.1 that these denotator values are visu-

[3] Uwe Laufs, Christopher Ruff, and Anette Weisbecker. "Mt4j: an open source platform for multi-touch software development". In: *VIMation Journal* (2010).

[4] For more information on the multi-touch version, see Thalmann and Mazzola, "Affine Musical Transformations Using Multi-touch Gestures", as well as forthcoming papers.

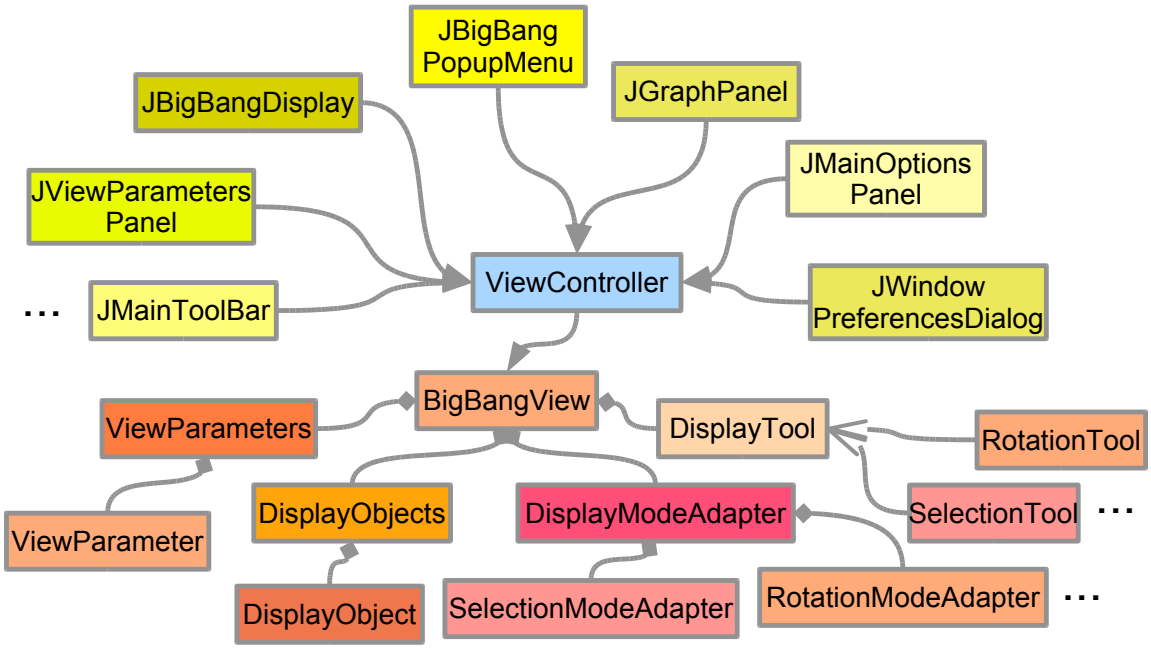


Figure 9.6: Some of the classes in *BigBang*'s view system (package *org.rubato .rubettes.bigbang.view*).

alized by being associated with a set of given visual parameters. In practice, this association happens in the `ViewParameters`class, where all possible denotator parameters values found in `BigBangObjects` are paired with the currently selected set of view parameters, currently including a choice of either hue-based or RGB-based colors (Section 6.3.3). `BigBangView` currently also manages its current user mode, which is represented by the subclasses of `DisplayModeAdapter` – which is technically a controller class, but is in this case also used to determine the mode in the view model – and the ones of `DisplayTool`. With these, each view has a current mode which determines how the user's gestures affect the composition. For instance, in *rotation* mode, in the mouse version of *BigBang*, a click defines the center of a rotation, whereas a click-and-drag determines its angle. While the user is performing gestures, the `RotationTool` is displayed as a referential representation of the current rotation.

9.3.2 The (Sub)View Classes and their Controller Classes

Many classes are views of *BigBang*'s view model, some of which are shown in Figure 9.6. The most important ones correspond to *BigBang*'s facts and process views (Chapter 5), `JBigBangDisplay` and `JGraphPanel`, respectively. Both of these views, as most other views in *BigBang* are contained in the class `JBigBangPanel`, ommited from Figure 9.6. The former visualizes the `DisplayObjects` using Java Swing graphics, whereas the latter provides an interactive visualization of the `BigBangOperationGraph`, making use of the *JUNG* framework's visualization capabilities (Section 9.2.1), which also based on Swing.

There are many other classes that correspond to various parts of the visual interface and offer ways of controlling *BigBang* through buttons, boxes, and menus. `JMainOptionsPanel` combines view parameters settings (the checkbox grid mentioned in Section 6.2.1), with settings that control on which level and which objects the user wants to draw, and other general settings such as a checkbox for whether *BigBang*'s input is active and receives denotators or not. Two toolbars, `JMainToolBar` and `JLayersToolBar` allow users to change *BigBang*'s mode, play back the composition, specify wallpaper and alteration settings, and navigate the layers, if any are defined. A popup menu, `JBigBangPopupMenu`, brought up by a right click on the facts view, allows an alternate access to certain functionality, such as building satellites, deleting objects, etc. Finally, the `BigBangView`'s more detailed settings menu is situated in `JWindowPreferencesDialog`, which can be brought up through the popup menu or a key command (ctrl-P or command-P) and for instance contains view parameter or synthesizer configuration settings.

All these views share a large number of actions and adapters, all located in the package `org.rubato.rubettes.bigbang.view.controller`, through which they can perform the activities available in *BigBang*. These actions and adapters simply call

methods of the `ViewController`, which in turn forwards the calls to the model classes. For instance, the `PlayButtonAction` calls `ViewController.togglePlayMode()`, which calls the same method in `BigBangView`, which in turn forwards the request to the `BigBangPlayer`.

9.4 Synthesizer and MIDI Classes

The last group of classes that we will examine a little closer here are the ones responsible for receiving MIDI input and playing back *BigBang*'s composition via Java synthesizer, Java MIDI, as well as MIDI. These classes are currently tied to the `BigBangView`, but due to their recent evolution, they will soon be relocated to a separate package. Even though the views show different perspectives, *BigBang* currently only contains one composition at a time, and thus only needs one set of sound and MIDI input and output classes. In fact, the audible playback can be considered a view in itself, that does however not need to be separated from the model through `BigBangController`. Figure 9.7 shows the current structure of *BigBang*'s sound and MIDI setup.

In the upper left of the figure, we find MIDI input and output classes, located in the package `org.rubato.rubettes.bigbang.view.io`, which make use of the standard `javax.sound.midi` package in order to obtain access to any `MidiDevice` currently connected to the system. `BigBangMidiReceiver` filters any MIDI message useful to *BigBang* and forwards it to the responsible instance. For example, all note on and note off messages trigger `pressMidiKey(...)` and `releaseMidiKey(...)` method calls in `ViewController`. Control change messages, in turn, are directly forwarded to `BigBangController.modifyOperation(...)`, as discussed in Section 8.1.4. The `BigBangMidiTransmitter` is used to play back MIDI and send MIDI messages to any

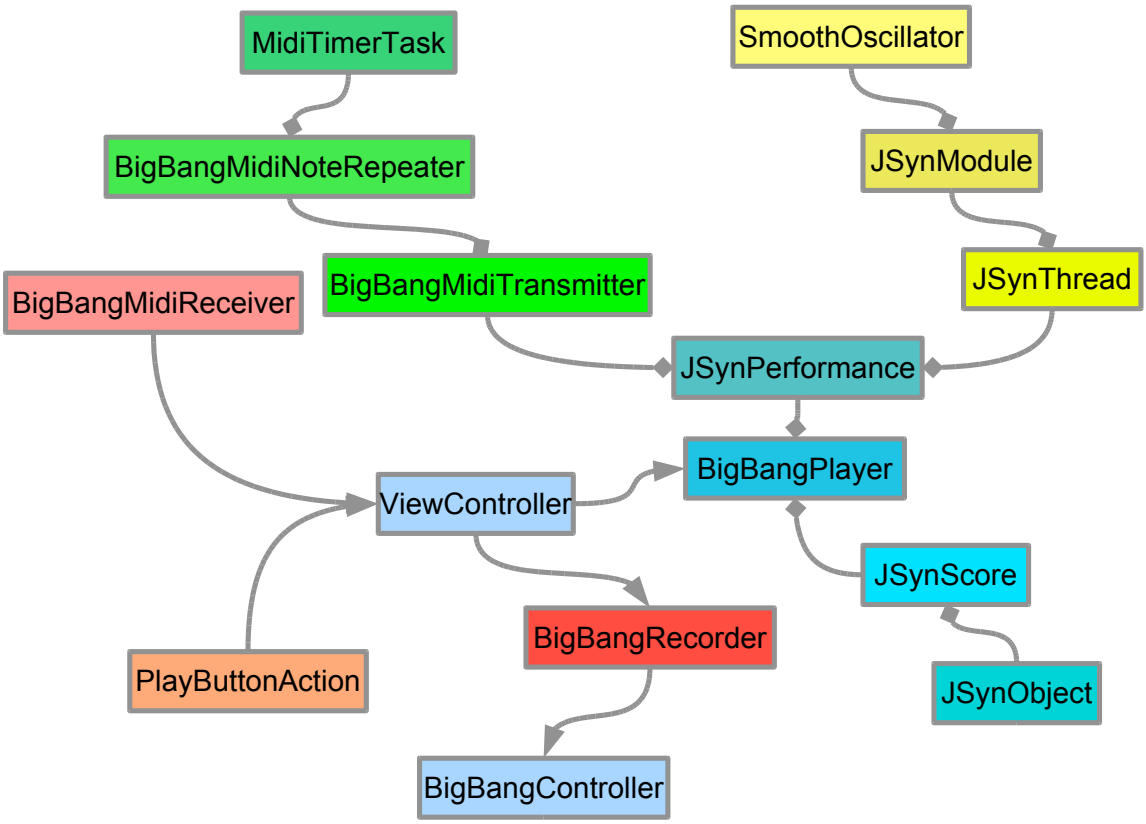


Figure 9.7: Some of the classes of *BigBang*'s sound and MIDI classes (packages *org.rubato.rubettes.bigbang.view.io* and *org.rubato.rubettes.bigbang.view .player*).

`MidiDevice`, currently only through `BigBangPlayer`'s playback mechanism.

`BigBangPlayer` is used to play back *BigBang*'s composition both by a microtonal synthesizer built with *JSyn*,[5] a versatile Java API that allows for a great variety of computer music applications and synthesizer programming, and via MIDI, using `javax.sound.midi`. `BigBangPlayer` is currently triggered through `ViewController` and `BigBangView`, for instance via a `PlayButtonAction`, as suggested in Figure 9.7, which toggles playback on or off. Also, whenever a change is made to *BigBang*'s composition, the `BigBangPlayer` is updated with the newest `BigBangObjects`, just as `BigBangView`. These objects are then converted into a playable sound object,

[5]Phil Burk. "JSyn – A Real-time Synthesis API for Java". In: *Proceedings of the International Computer Music Conference*. San Francisco: International Computer Music Association, 1998.

currently called `JSynScore`, a set of sounding `JSynObjects`. As described in Section 6.4, any audible dimension not present in the object is automatically completed, so that almost any denotator can be sonified. The `JSynScore` is then performed by `JSynPerformances`, of which several can happen simultaneously, for instance when the user is triggering playbacks using a MIDI controller (as described in Section 6.4). A `JSynPerformance`, in turn, consists of several `JSynThreads`, depending on the number of simultaneous voices, which are all associated with one `JSynModule` containing a number of `SmoothOscillators`, an oscillator specifically made for *BigBang*, which automatically sweeps both amplitude and frequency, in order to avoid artifact clicks. `SmoothOscillators` can also be combined into greater structures, using the modular `SmoothOscillatorModules`, with which one can easily build components for frequency modulation, ring modulation, and additive synthesis.

Instead of playing back `JSynScores` as synthesizer objects, `JSynPerformances` can also send corresponding messages to the `BigBangMidiTransmitter`, which then converts them to appropriate MIDI messages and sends them to MIDI devices, including the MIDI module of the Java Sound Synthesizer. Both ways of playing back can also happen at the same time.

9.5 Other Implementation Details

A few more aspects of the implementation are relevant in the context of this thesis and will thus briefly be summarized. First, according to the design principles underlying *Rubato Composer*, every rubette needs to be able to be duplicated as well as saved along with the network it appears in. Programmers have a choice of what characteristics of the original rubette reappear in the duplicated or loaded versions. This does currently not work for all desirable parameters, but it will be adjusted in further

developments in the near future. The main workings, however, will be explained in this section. Furthermore, I will address a central part of the development process of *BigBang*: the classes responsible for automated testing.

9.5.1 Duplicating

Every rubette has to implement the `duplicate()` method, except for when it is based on `SimpleAbstractRubette`, where all properties are managed automatically. The method returns a new rubette that preferably maintains the same properties and other settings as the original. It is used, for instance, when users copy and paste a rubette in the *Rubato Composer* network environment.

With *BigBang*, we expect a lot to be copied along, foremost its composition process, but possibly also some of its view settings, etc. For now, the `duplicate()` method simply looks as follows:

```
public Rubette duplicate() {
        return new BigBangRubette(this.model.clone());
}
```

It uses *BigBang*'s private constructor that accepts an already existing `BigBangModel` as an argument. As can be seen in the code, `BigBangModel` offers a `clone()` method, which creates a deep copy of the model including all of its components: the operation graph, the denotators, and the `BigBangObjects`. In fact, the only thing that needs to be copied is the form as well as the operation graph and all of its operations, while the denotators and the `BigBangObjects` can be dynamically generated from the graph, with a call of the `updateComposition()` method, as described in Section 9.2. The model's `clone()` method is thus:

```
public BigBangModel clone() {
        BigBangModel clonedModel = new BigBangModel();
        clonedModel.setForm(this.denotators.getForm());
        clonedModel.operationGraph = this.operationGraph.clone(clonedModel);
        clonedModel.updateComposition();
        return clonedModel;
}
```

The limitations of this method of duplicating is that the `BigBangObjects` are not properly duplicated, which makes it impossible to duplicate operations that use specific selections of objects. Instead, at the current stage of implementation, all operations are duplicated without their references to objects. This has the consequence that all operations of the duplicated rubette are always applied to all objects existing at their composition state, just as described in Section 7.1.1. However, the method thus works well for abstract operation graphs, designed independently of any compositional objects, as well as graphs where operations always concern all objects. A consequential limitation of this method is also that duplicating does currently not work for all operations. Alterations, for example, do not make sense if their two compositions are identical and both are the entire composition. These limitations will be dealt with in the near future, with the addition of the ability to duplicate `BigBangObjects` and their references in operations.

9.5.2 Saving and Loading

In addition to duplicating, rubettes also need to be able to be saved and loaded along with their networks. *Rubato Composer* saves any of the data created by the user at runtime, including modules, forms, denotators, rubettes, and networks, into *XML* files. Each rubette must implement a `toXML(...)` method for saving and a `fromXML(...)` for loading. A few classes that are part of *Rubato Composer*'s framework simplify this process, such as for instance `XMLWriter` and `XMLReader`, which

automatically create and parse XML tags.

We expect *BigBang* to be able to be saved and loaded in exactly the same way as it can be duplicated, and we thus encounter similar problems. In the current version, only the operation graph and thus only operations that are not based on selections of `BigBangObjects` can be saved. *BigBang*'s methods look as follows:

```
public void toXML(XMLWriter writer) {
        this.model.toXML(writer);
}

public Rubette fromXML(XMLReader reader, Element element) {
        BigBangModel loadedModel = BigBangModel.fromXML(reader, element);
        return new BigBangRubette(loadedModel);
}
```

Again, the only saved instance is *BigBang*'s model, which offers corresponding XML methods. The model's `toXML(...)` methods is simply

```
public void toXML(XMLWriter writer) {
        writer.writeFormRef(this.denotators.getForm());
        this.operationGraph.toXML(writer);
}
```

analogous to the `duplicate()` method, saving only the form and the operation graph. Along with using the standard *Rubato Composer* XML tags, such as for instance for the form, many new XML tags had to be defined in order to save all of the operation graph's aspects. For example, the main graph tag *OperationGraph* has one integer attribute *numberOfStates*, and contains a number of *Operation* tags, each of them referring to an *operationName* and a *head* and *tail* state, among other things.

The model's `fromXML(...)` method works accordingly:

```
public static BigBangModel fromXML(XMLReader reader, Element element) {
        BigBangModel loadedModel = new BigBangModel();
        try {
                Form form = reader.parseAndResolveForm(XMLReader
                        .getChild(element, FORM));
                loadedModel.setForm(form);
                loadedModel.setGraph(BigBangOperationGraph
                        .fromXML(loadedModel, reader, element));
                loadedModel.updateComposition();
        } catch (Exception e) {
                e.printStackTrace();
        }
        return loadedModel;
}
```

Again, the `updateComposition()` method is called after the form and the graph are loaded, in order to create denotators and `BigBangObjects`.

9.5.3 Testing BigBang

When developing a software component as complex as the *BigBang* rubette, it is crucial to constantly test as many as possible of the ways it can be used in. Even though the ultimate use of the *BigBang* rubette is gestural interaction through its visual interface, which is difficult to be tested non-manually, much of its basic functionality could be verified systematically without *Rubato Composer* having to be started, and without tedious and repetitive user interaction. For this, the code contains *JUnit* test classes with extensive `TestCases`, situated in the package `org.rubato.rubettes.bigbang.test`. Especially the generality of *Rubato Composer* and *BigBang*, with the possibility to create an infinite number of mathematical structural types, is challenging in the context of testing. Thus, most of the test classes work with a number of exemplary forms, each of them with different characteristics. The following list compiles all current test classes:

```
BigBangOperationGraphTest.java
UndoRedoTest.java

ArbitraryDenotatorMapperTest.java
BigBangDenotatorManagerTest.java
DenotatorPathTest.java
ObjectGeneratorTest.java
PowerDenotatorTest.java

BigBangObjectsTest.java
DenotatorValueExtractorTest.java
FormValueFinderTest.java

DisplayObjectsTest.java
JSynPlayerTest.java
ViewParameterTest.java

TestObjects.java
```

The above tests are divided in a few groups for clarity. The first group tests the operation graph and the undo/redo part of the model, the second group the classes that deal with denotators, and the third the model classes related to the `BigBangObjects`. The fourth group deals with what is testable in the view and player classes. Finally, the last class, `TestObjects.java`, contains many standard denotators and procedures available to all test classes. The next code excerpt shows a sample test taken from `UndoRedoTest.java`:

```
public void testUndoRedoParallelTransformation() {
        //add score
        this.model.setOrAddComposition(this.objects.flatSoundScore);
        this.checkGraphAndResult(2, 1, new double[][]{
                {0,60,120,1,0},{1,63,116,1,0},{2,60,121,1,1}});

        //translate
        this.addOnsetPitchTranslation(-1, 2);
        this.checkGraphAndResult(3, 2, new double[][]{
                {-1,62,120,1,0},{0,65,116,1,0},{2,60,121,1,1}});

        //parallel translate
```

```
        this.model.selectOperation(this.model.getTransformationGraph()
                .getLastAddedOperation());
        this.addOnsetPitchTranslation(-1, 2);
        this.checkGraphAndResult(3, 3, new double[][]{
                {-2,64,120,1,0},{-1,67,116,1,0},{2,60,121,1,1}});

        //add sequential translate
        this.model.selectOperation(null);
        this.addOnsetPitchTranslation(2, -1);
        this.checkGraphAndResult(4, 4, new double[][]{
                {0,63,120,1,0},{1,66,116,1,0},{2,60,121,1,1}});

        //undo sequential translate
        this.model.undo();
        this.checkGraphAndResult(3, 3, new double[][]{
                {-2,64,120,1,0},{-1,67,116,1,0},{2,60,121,1,1}});

        //undo parallel translate
        this.model.undo();
        this.checkGraphAndResult(3, 2, new double[][]{
                {-1,62,120,1,0},{0,65,116,1,0},{2,60,121,1,1}});

        //undo parallel translate
        this.model.redo();
        this.checkGraphAndResult(3, 3, new double[][]{
                {-2,64,120,1,0},{-1,67,116,1,0},{2,60,121,1,1}});

        //redo sequential translate
        this.model.redo();
        this.checkGraphAndResult(4, 4, new double[][]{
                {0,63,120,1,0},{1,66,116,1,0},{2,60,121,1,1}});
}
```

This test verifies the undo/redo model's behavior upon the special case of parallel and sequential transformations. It makes use of the `TestObjects`' predefined `flatSoundScore` (containing no satellites) and step by step adds operations and transformations, then undoes, and finally redoes them. At each step of the procedure, the code verifies the result using a simple private method:

```
private void checkGraphAndResult(int expectedStates,
```

```
                int expectedOperations, double[][] expectedValues) {
        TestCase.assertEquals(expectedStates,
                this.model.getTransformationGraph().getVertexCount());
        TestCase.assertEquals(expectedOperations,
                this.model.getTransformationGraph().getEdgeCount());
        Denotator expectedResult = this.objects.generator
                .createFlatSoundScore(expectedValues);
        TestCase.assertEquals(expectedResult, this.model.getComposition());
}
```

In a similar way, many exemplary situations and special cases among *BigBang*'s possibilies are tested, and at each step of the implementation they can be run, in order to help the developer make sure that all of the previously implemented features are still functional.

Chapter 10

Musical Examples

The new *BigBang* rubette offers many possibilities of creating music, due to the great variety of forms that can be defined. I already presented some simple ideas of forms in Section 6.5. In this section, I introduce some of innumerable slightly larger musical examples created in the course of writing the code of *BigBang* and this thesis. These examples illustrate a variety of composition techniques and types of musical results possible with *BigBang*. All examples are available for listening on SoundCloud, and some of the more performative ones can be found on YouTube, under the addresses indicated below.

10.1 Some Example Compositions

This section explores some of the compositional possibilities of *BigBang*, i.e. preparing music outside of musical time that can later be played back, recorded, or performed, as discussed in Section 4.1. More spontaneous and real-time ways of creating music with *BigBang* will be discussed in the next section.

10.1.1 Transforming an Existing Composition

Form *Score*

Graph 4 states, 3 sequential operations

Techniques inputing a composition, transforming, modifying

Output *BigBang* synth with sine wave oscillators, slightly post-processed

Link `http://www.soundcloud.com/bigbangrubette/k003`

Instead of creating denotators from scratch, there are many ways in which existing composition can be used to create strikingly different musical results. This example is part of a series of variations based on Sonatas by Domenico Scarlatti, all of them using composition procedures that are as simple as possible. Here, I input Scarlatti's *K003* into *BigBang* via a *MidiFileIn* rubette, thus in *Score* form, then I stretched and compressed it in time and pitch, respectively (`ScalingTransformation`), and finally transposed it down several octaves (`TranslationTransformation`). The resulting graph therefore consists of three sequential operations (see Figure 10.1). Using the option of modifying transformations, I found the range I envisioned, resulting in a pulsating bass sound emerging from the beating based on the close frequencies after the pitch compression. The clicking noise, resulting from a chosen short attack time of the *BigBang* synthesizer, preserves the rhythmical qualities of the Scarlatti. The visualization of the final result (Figure 10.1) was partially created due to aesthetic decisions. However, it shows the composition on the $Onset \times Pitch$ plane, where the close *Pitch* range is visible (the vertical middle of the blocks), around MIDI pitch 24, which corresponds to $C1$ or approximately 32 Hz.

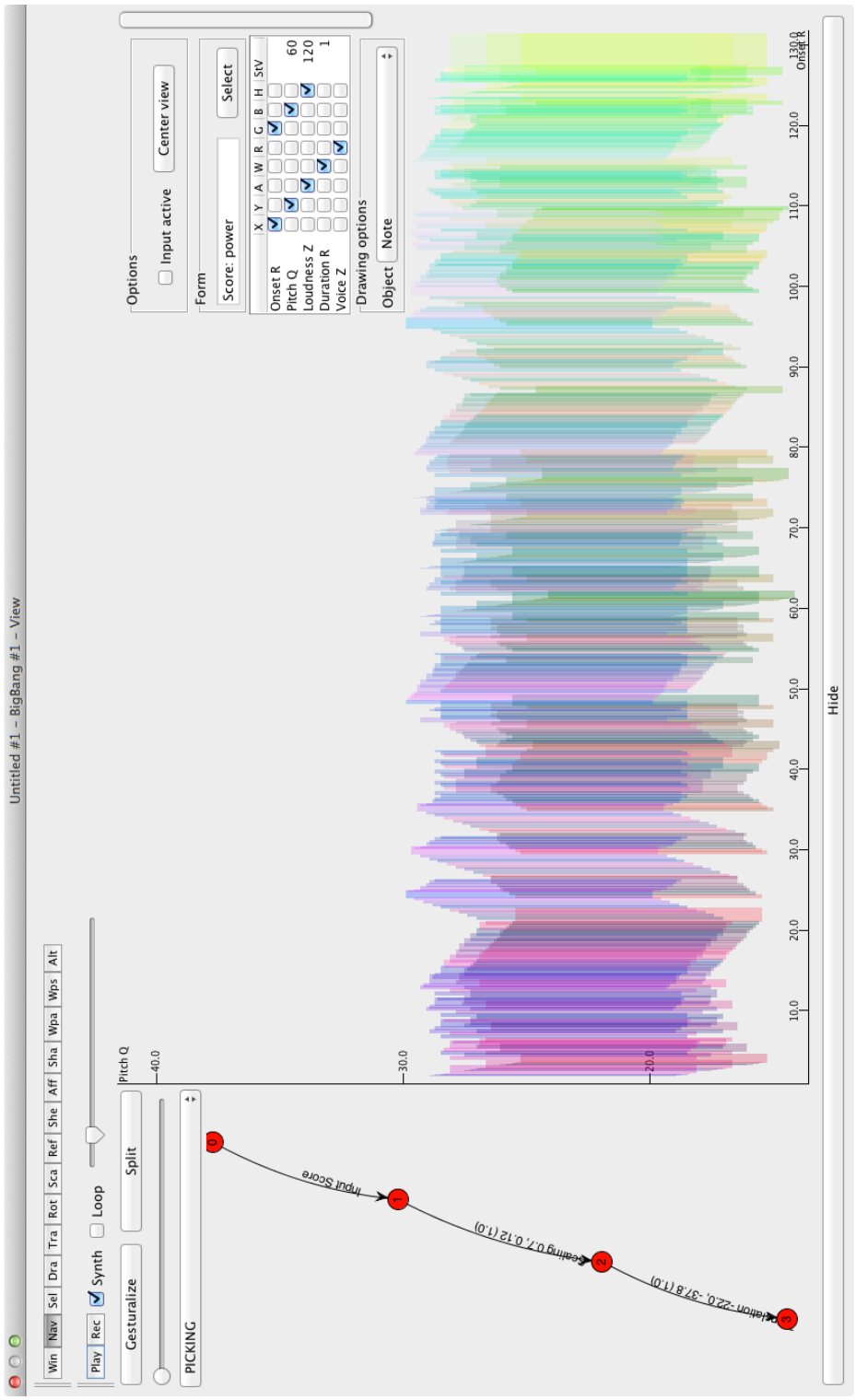


Figure 10.1: A transformation of Scarlatti's Sonata *K003* resulting in a pulsating bass sound.

10.1.2 Gesturalizating and Looping with a Simple Graph

Form *Score*

Graph 2 states, 2 parallel operations

Techniques looping, gesturalizing

Output MIDI to Ableton Live, Guitar-Jazz preset

Link `http://www.soundcloud.com/bigbangrubette/k002`

Another piece part of the Scarlatti series, this example uses *BigBang*'s gesturalizing function. Its graph consists of merely two states, between which we find two parallel operations. Again, the original (*K002*) enters through a *MidiFileIn* rubette, resulting in a `InputCompositionOperation`. Then, by selecting the operation and performing a counterclockwise rotation by 180 degrees, I added a parallel `RotationTransformation`, which results in a minimal graph with two states and two operations (Figure 10.2). When gesturalized, these two operations occur simultaneously (see Section 7.3.3), so that the composition simultaneously grows note by note, and gradually rotates. During gesturalization, the composition is played back in loop mode, where in this case, the loop grows longer and longer, and output through MIDI directly to Ableton Live, where it is played back by a guitar sound. In order to find a musical result, I experimented with operation durations and tempo, settling on a gesturalization time of 200 seconds at a pace around two to three times as fast as the tempo the sonata is often played at. The resulting piece has a strong improvisational and gestural quality, where the motivic content is gradually developed and grows larger and larger. A contrapuntal effect reminiscent of group improvisation emerges due to the variety of pitch ranges produced by the counterclockwise rotation, which are well captured by lower end of the guitar sound. In the end, the musical material

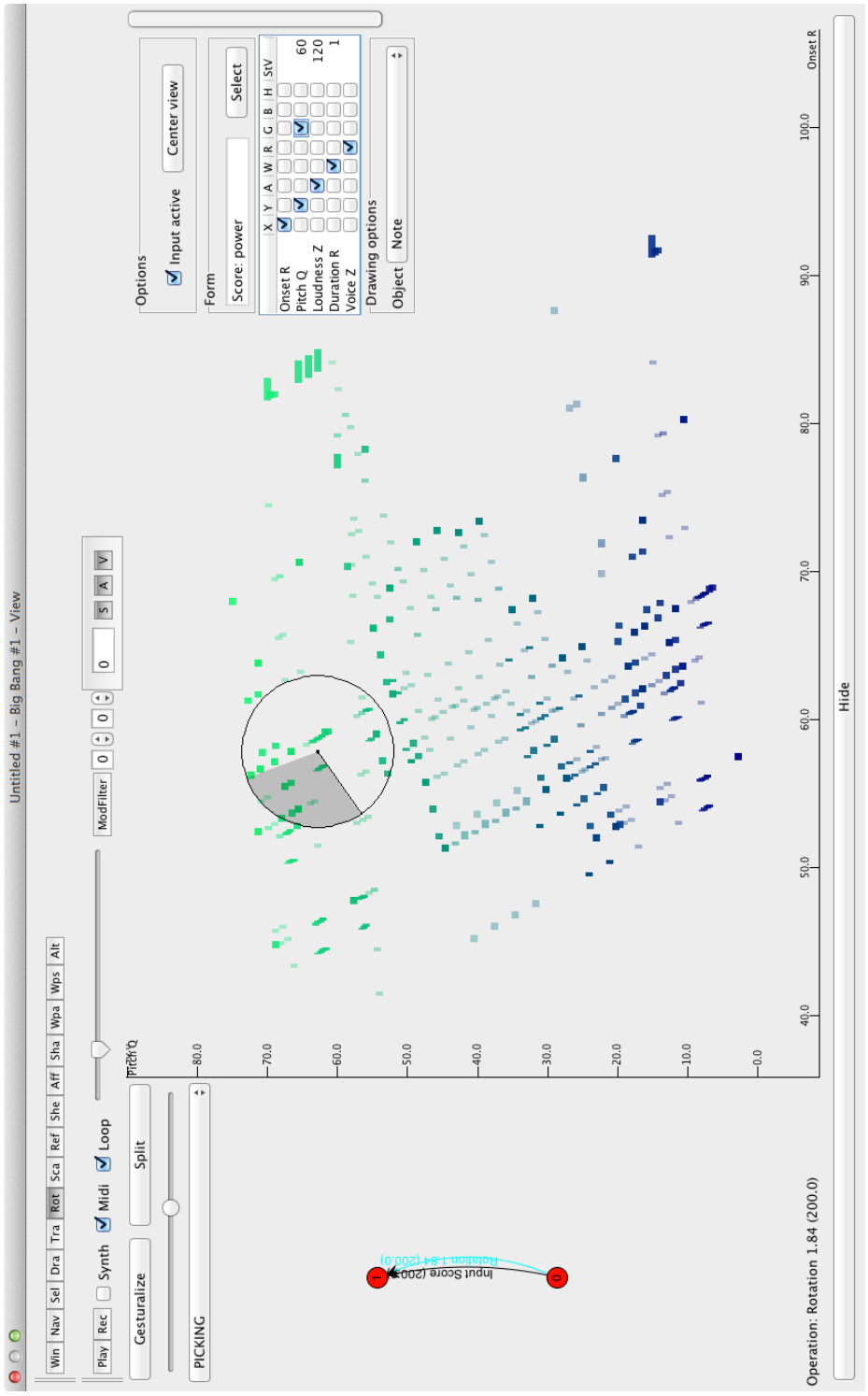


Figure 10.2: A growing and rotating Scarlatti *K002* during gesturalization.

converges towards the retrograde inversion of the Scarlatti, modulating increasingly slowly, and culminating in a congenial closure. Figure 10.2 shows the piece shortly after midway through the gesturalization.

10.1.3 Drawing UPIC-Like Motives and Transforming

Form *PanScore*

Graph long sequential graph

Techniques drawing, shaping, copy-and-transforming

Output *BigBang* synth with sine wave oscillators, post-processed

Link `http://www.soundcloud.com/bigbangrubette/upic`

In comparison with Xenakis's *UPIC* system, discussed in Section 4.3.1, *BigBang* has several advantages, the two most important of which are that composers can work with arbitrary musical object types, and that they can transform these objects. This example makes use of the latter, while keeping a similar data type as was used with Xenakis's system, *Score*, however, an enhanced version that allows for stereo panning, which I called *PanScore*. The example is based on a drawn structure with ramifications similar to, for instance, parts of Xenakis's *Mycenae Alpha*. Since drawing can only occur in two dimensions at a time, I used shearing transformations as well as the shaping operation to process the drawn structure in dimensions other than *Onset* and *Pitch*, here mainly *Pan*. Then, I multiplied the initial motive and partial motives by using various copy-and-transform operations (Section 7.2.2), a simple and intuitive way of ensuring motivic unity in a piece. Figure 10.3 shows the score on the UPIC-typical $Onset \times Pitch$ plane. Figure 10.4 shows the results of the shaping and shearing on the $Onset \times Pan$ plane, with the same color distribution as

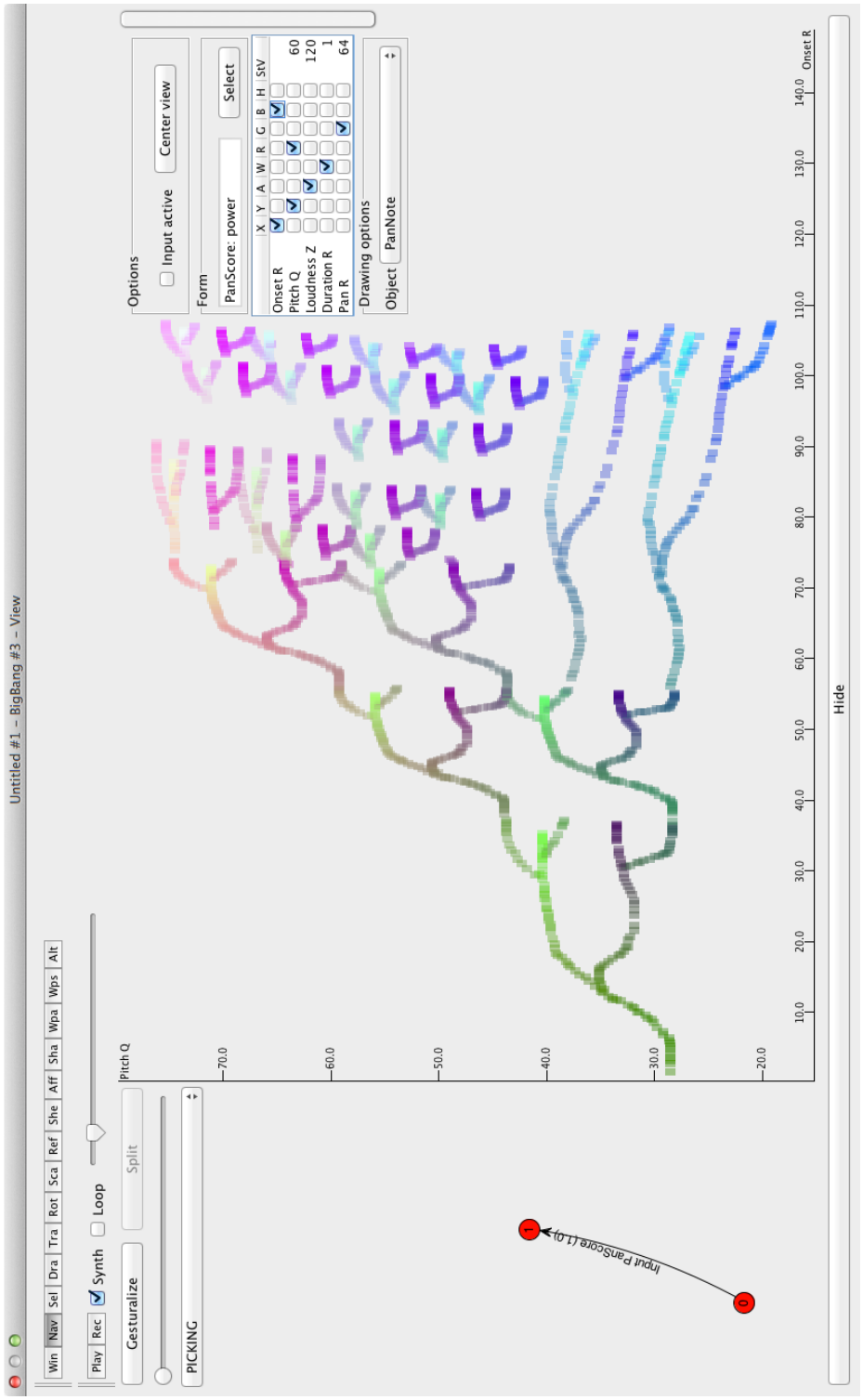


Figure 10.3: The *Onset* × *Pitch* plane of the *UPIC*-like composition.

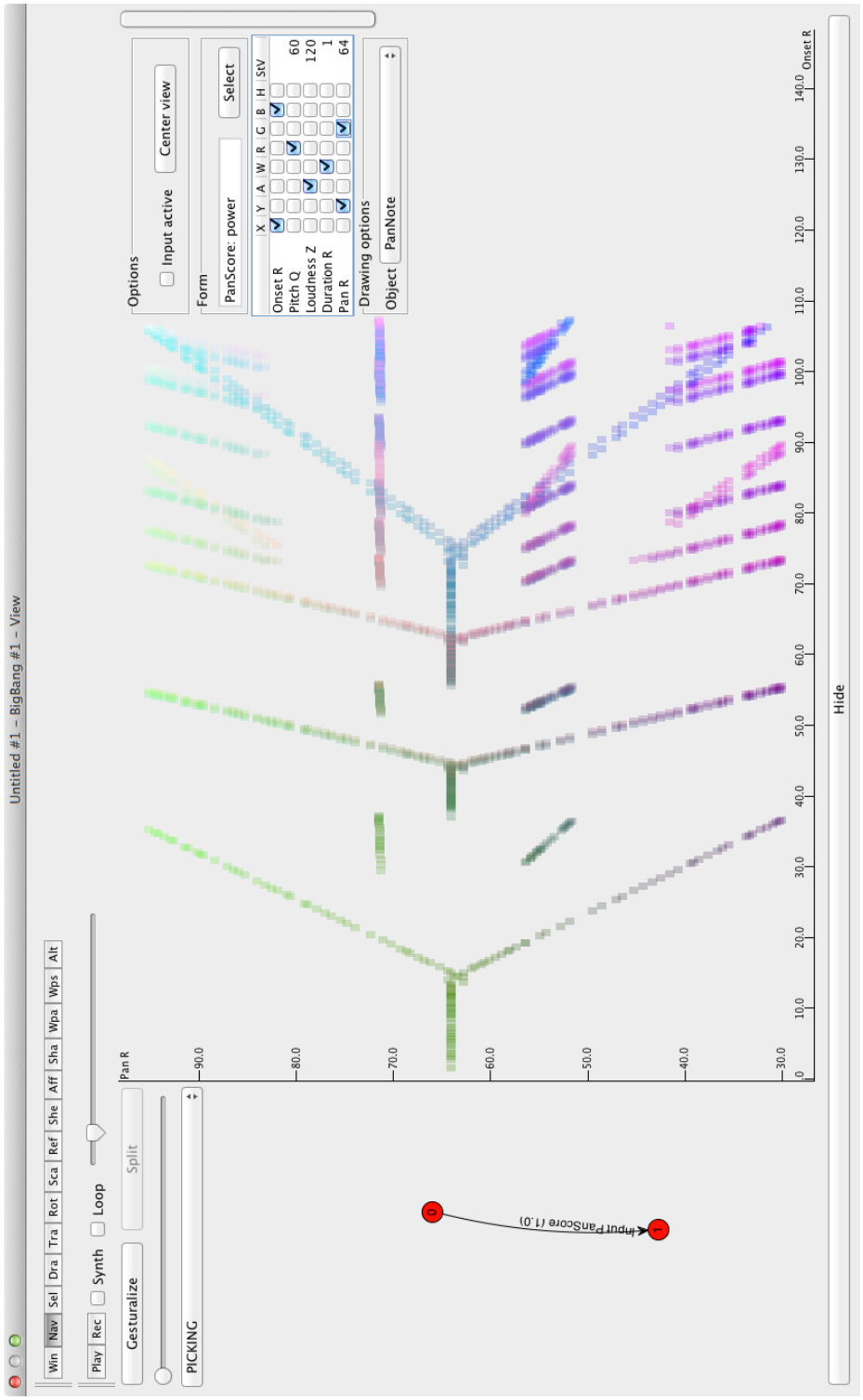


Figure 10.4: The *Onset* × *Pan* plane of the *UPIC*-like composition.

in the Figure 10.3. Note that the images do not contain the original graph, which was of linear nature and almost confusingly long, for the first result was saved (using the *Register* and *Source* rubettes) and worked on in several sessions. However, the original graph was of a linear nature. The result is a microtonal spectral composition that reiterates initial motive in increasingly contracted and cut out versions, evolving from a single voice to about 45. The result was post-processed in Ableton Live, with some reverb, equalizing, and compression.

10.1.4 Drawing Time-Slices

Form *PanScore*

Graph just a drawing operation

Technique drawing, changing preset values

Output *BigBang* synthesizer with triangle waves, post-processed

Link `http://www.soundcloud.com/bigbangrubette/slices`

This example illustrates another technique of drawing in several dimensions. Instead of switching to other planes and shaping and transforming drawn motives, as described in the previous example, it is also possible to determine the values in the dimensions absent from the x/y plane by entering standard values into the boxes to the right of each denotator value row in the view parameters table (right hand side of *BigBang* interface). In this case, I drew a *PanScore* on the $Pan \times Pitch$ plane, while manually entering *Onsets* and *Durations*. This way, starting with $Onset = 0$ and $Duration = 2$, I was able to draw overlapping slices of the same duration, by increasing *Onset* step by step, and by drawing increasingly many *PanNotes* in approximate concentric circles. For the second part of the piece, I drew single longer pitches, all of

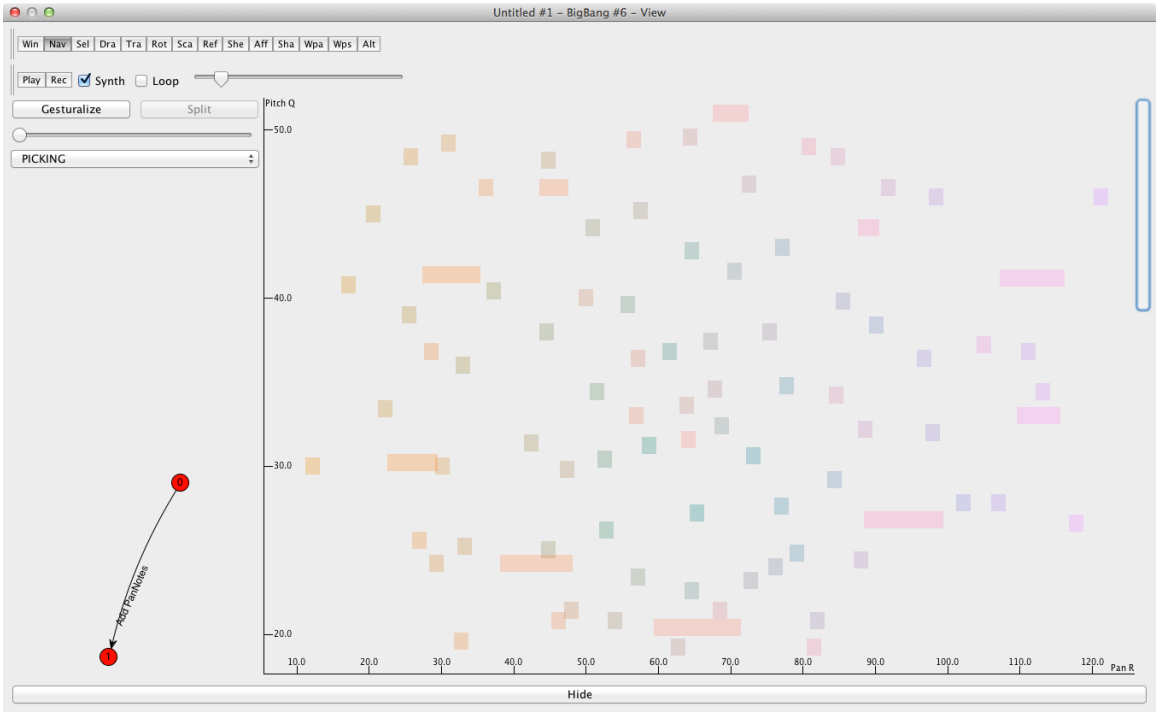


Figure 10.5: The *Pan* × *Pitch* plane on which drawing took place.

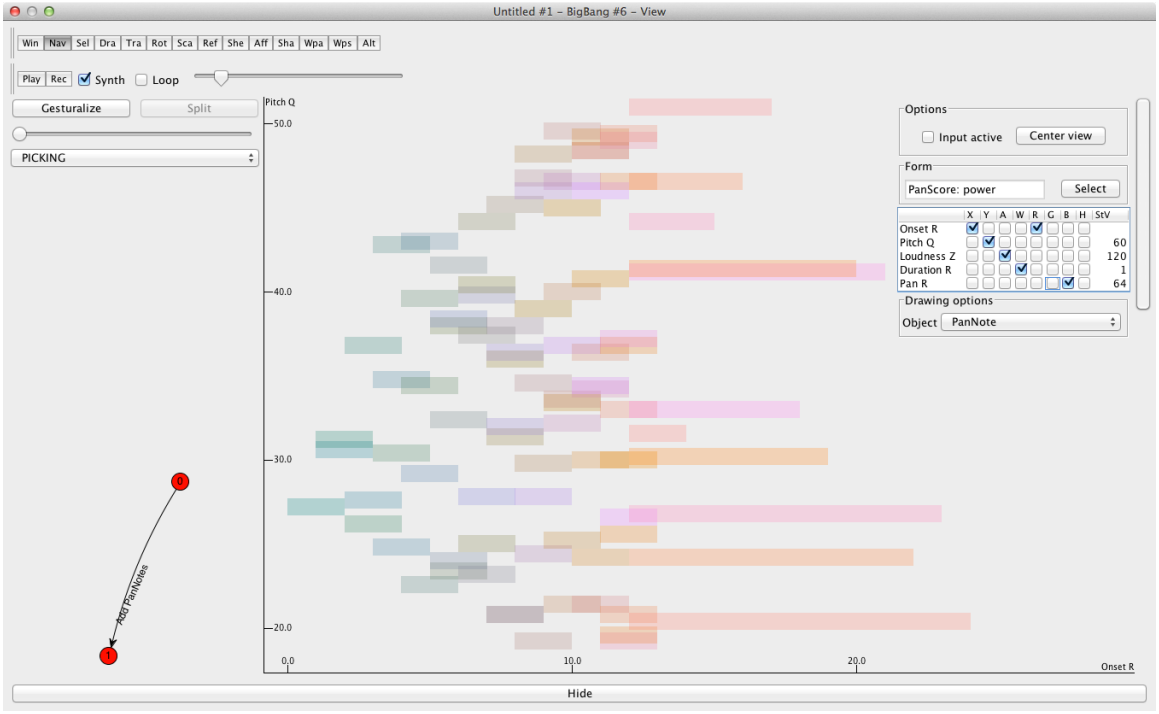


Figure 10.6: The resulting slices seen on the *Onset* × *Pitch* plane.

them at $Onset = 12$, first with $Duration = 12$, then decreasing the duration step by step, which resulted in a gradual disappearing of the pitches of the final chord. The piece is played back using triangle wave oscillators and mastered in Live. Figure 10.5 shows the drawing plane, whereas Figure 10.6 shows the resulting slices in a temporal representation, the colors being kept the same for both representations, in order to show the respective missing spacial dimension.

10.1.5 Converting Forms, Tricks for Gesturalizing

Form *Texture*

Graph several sequential and parallel operations

Technique reforming, identities, immediate operations, gesturalizing, selecting states

Output MIDI to Ableton Live, string ensemble

Link `http://www.soundcloud.com/bigbangrubette/textures`

This example is slightly more complex. It uses a network with three rubettes, one of them specifically created during the work on this thesis. The *Texturalize* rubette converts a *Score* into a *Texture* (introduced in Section 6.5.2) based on analytical values. It gathers all pitches present in the *Score* and for each *Pitch* p, remembers the number of occurrences o_p, the average duration d_p, as well as the average loudness l_p. The output of the *Texturalize* rubette is then a *Texture* with a *RepeatedNote* for each p, with a *Rate* based on o_p, a *Duration* based on d_p. and a *Loudness* based on l_p. The *Texture* can thus be seen as a scrambled but regularized version of the input piece, with the same average tone material, resembling the textures of the American Minimalists.

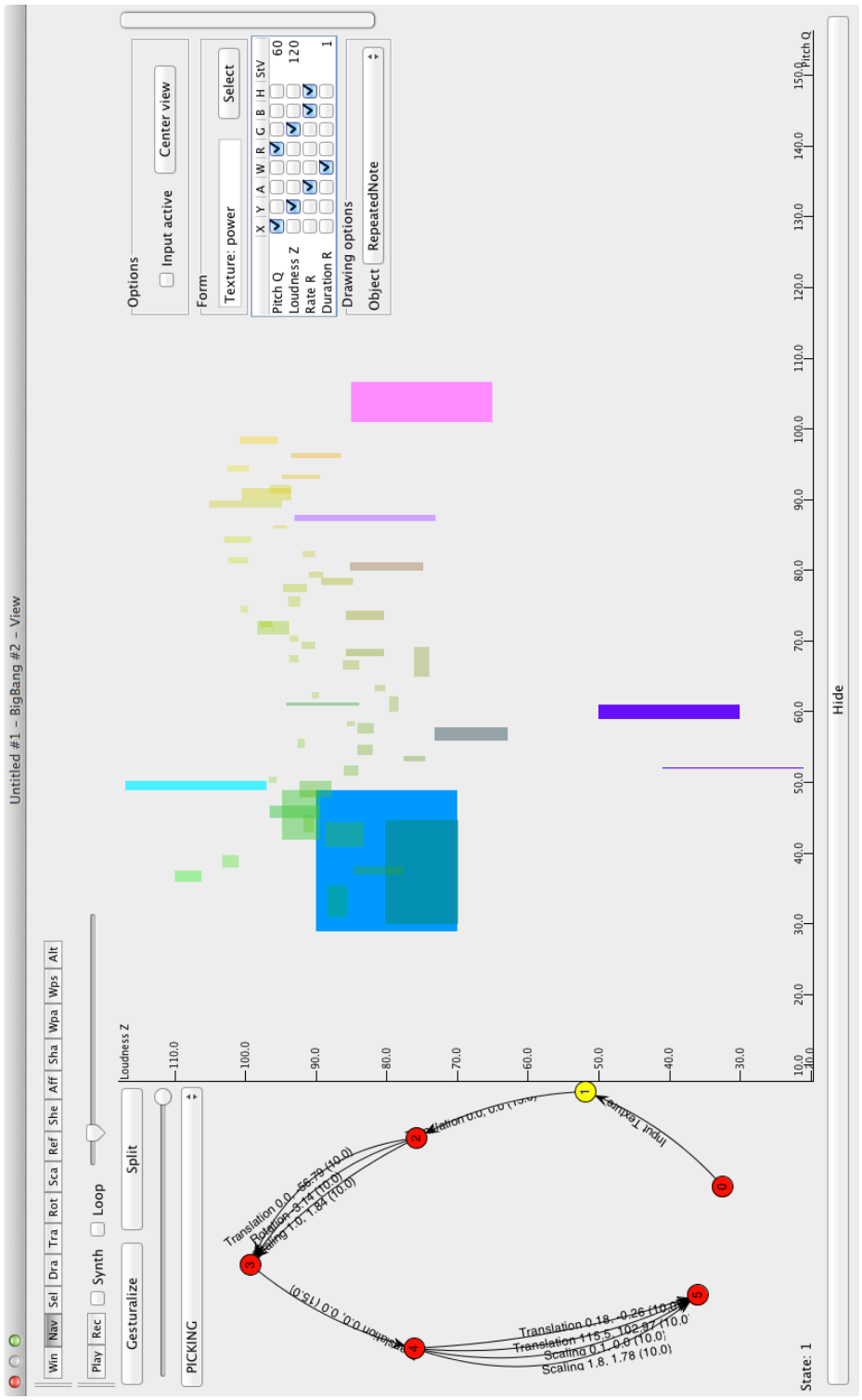


Figure 10.7: The facts view shows the *Texture* at state 1, with rate and duration represented by height and width, respectively. The process view shows the graph generating the entire composition.

The current example makes use of a *Texture* based on a part of a live performance of Chopin's *Ballade Op. 23* in *g* minor, at the indication *agitato* and then *sempre più mosso*. The fact that it is a live performance leads to a great variety of durations and dynamic values, as opposed to the notated score, which is particularly interesting when converted into a *Texture*. Figure 10.7 shows the facts at the initial stage (composition state 1).

The first part of the example, played by a string ensemble, is based on a gesturalization of various transformations of the original texture, which results in slowly changing rates, durations, loudnesses, and pitches of the texture's notes. In addition to the evolving parts, I also wanted to include parts where the current texture is resting. Currently, the trick to do this is to insert an identity transformation, for instance, a translation by 0, as in the example, and assign the transformation a gesturalization duration. Another trick is used in the beginning of the piece, where I did not want the texture to gradually build up when gesturalized. For this, I assigned the `InputCompositionOperation` a duration of 0.

The entire trajectory of the first part reaches three stable states, one at the original texture, one at a lower, quieter, and more legato retrograde inversion of the original, and one at a variation that is louder, faster, staccato, and that consists of an extended pitch space. These variations of the original texture come about using parallel transformations, which are equally gesturalized, with the effect of two gradual textural changes between the three static parts. The first transition consists of a translation down in loudness, a rotation by 180 degrees on the loudness/pitch plane, and a scaling in duration in order to make the notes longer. The second transition consists of a translation up in loudness, a scaling and a translation on the rate/duration plane, and a scaling on the loudness/pitch plane in order to expand both dynamic range and pitch range. At the end of the first part, the piece jumps back to the original

texture, which was done manually by selecting composition state 1.

The second part of the piece, realized by pizzicato strings, makes use of a technique that rather fits into the part on improvisation and performance with *BigBang*. Using the same graph, I recorded the strings playing while I jumped from composition state to composition state, using the number keys of the computer keyboards. This way, the strings freely jump back and forth between the three textural states and remain static for various amounts of time.

10.1.6 Gesturalizing a Spectrum

Form *Spectrum*

Graph many sequential and parallel transformations

Technique drawing, selecting and transforming groups of objects

Output *BigBang* synthesizer with sine waves, post-processed with ring modulation

Links `http://www.soundcloud.com/bigbangrubette/spectrum4`

`http://youtu.be/JlIpjOlKYUc`

Gesturalization was already illustrated in the examples in both Sections 10.1.2 and 10.1.5. However, the two forms used there, *Score* and *Texture*, define denotators with a distinct temporal existence, through the **Simple** forms *Onset*, *Duration*, and *Rate*. As described in Section 6.4, if none of these forms are present, the musical objects sound constantly when played back. This is especially interesting when they are gesturalized, which results in microtonal glissandi. In this example, I used the *Spectrum* form to create a slowly evolving spectral texture. The gestural result starts out by gradually adding *Partials*, in the order I defined them. After that, the *Partials* are transformed either sequentially or in parallel, in pairs or as a whole, resulting from

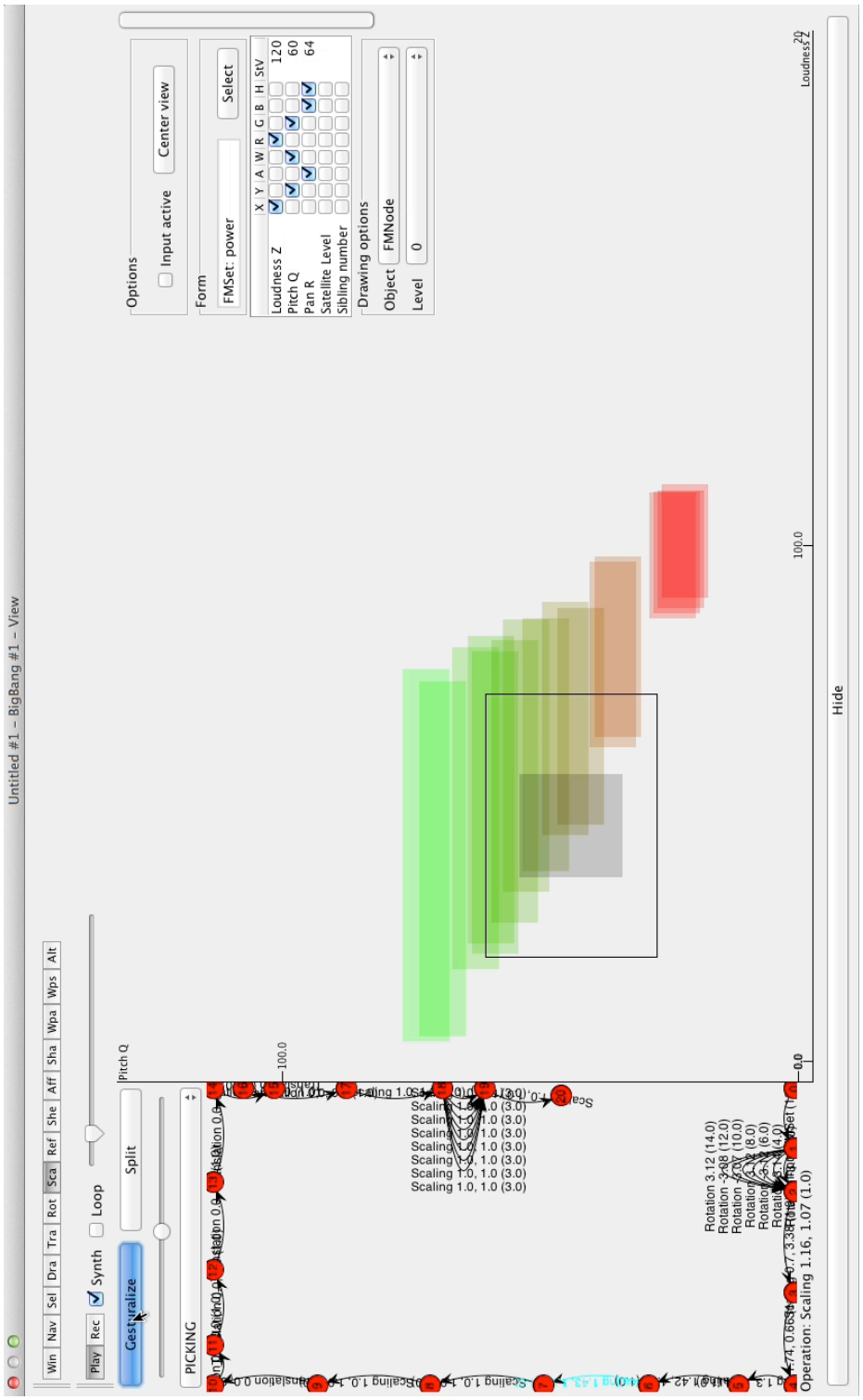


Figure 10.8: The *Spectrum* during gesturalization. For a video, see the link above.

differently timed transformations based on various selections of *Partials*. Figure 10.8 shows a moment during one of the sequential scalings of a pair.

10.1.7 Using Wallpapers to Create Rhythmical Structures

Form *Score*

Graph a wallpaper with a few sequential transformations

Technique creating regular structures using wallpapers

Output MIDI to GarageBand

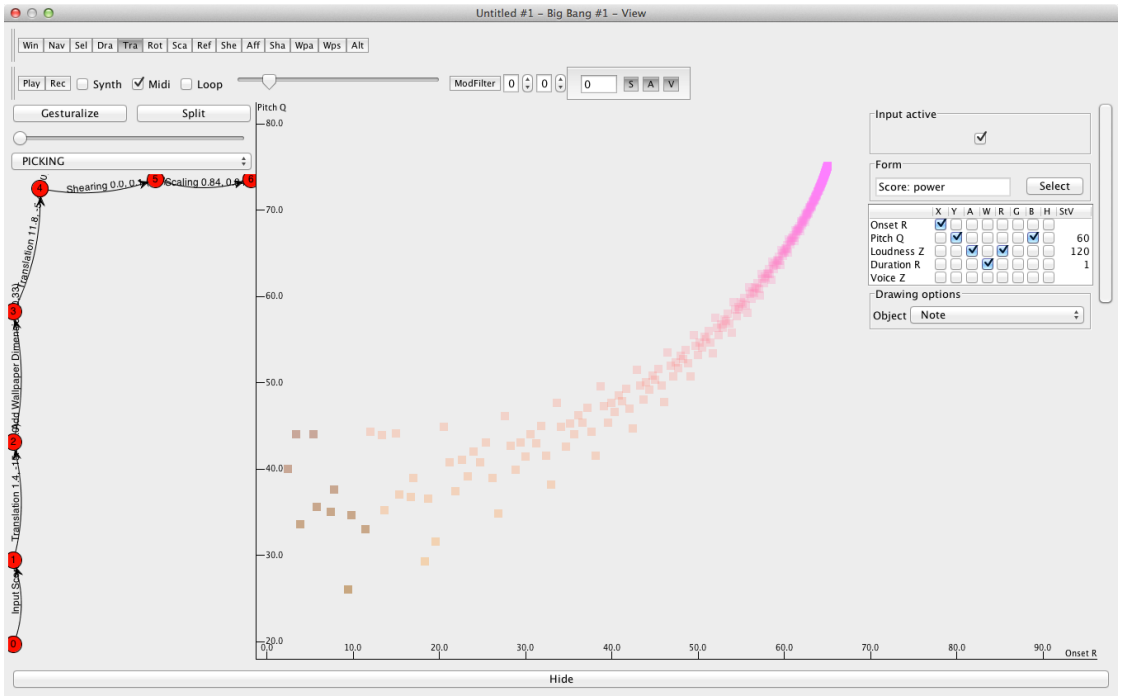
Links `http://www.soundcloud.com/bigbangrubette/wallpapers`

One of the main uses of ornamental structures made with the *OrnaMagic* module in the *Presto* software, following Mazzola, was the creation of regular drum patterns. Mazzola's *Synthesis* composition[1] makes wide uses of transformationally reiterating drum patterns created this way. With *BigBang*, such structures can be created in a much less tedious way, as this small example shows. Two short random drum motives are created using the *Melody* rubette, input into *BigBang*, and iteratively transformed with two different wallpapers, as shown in Figure 10.9. This way, we obtain slowly altering drum patterns, the first one translated, sheared, and scaled, and the second one translated and scaled.

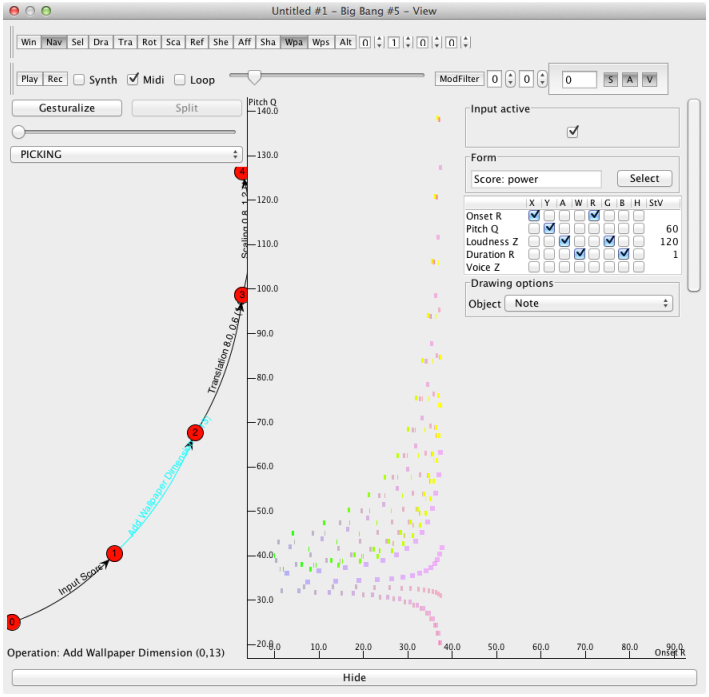
10.2 Improvisation and Performance with BigBang

Even though many of the aspects of the compositional examples seen so far were created in spontaneous ways, we now turn to a discussion of more momentary ways of

[1] Guerino Mazzola. *Synthesis*. Zürich: SToA music CD ST-71.1001, 1990.



(a)



(b)

Figure 10.9: The two wallpaper patterns in this example.

creating music with *BigBang*. Several aspects of the requirements for gestural control, discussed in Section 4.3.3 are ideal for more performative and improvisatory ways of making music. For instance, the fact that at all times, when working with *BigBang*, there is immediate auditory and visual feedback inspires musicians to experiment and spontaneously react to the outcome of their actions. Also, the support of various physical interfaces enable users to handle *BigBang* like an instrument.

10.2.1 Improvising by Selecting States and Modifying Transformations

Form *Spectrum*

Graph sequential and alternative transformations

Technique selecting states with number keys, modifying transformations with MIDI controller knobs

Output *BigBang* synth with sine waves, postprocessed

Link `http://www.soundcloud.com/bigbangrubette/selections`

The first example simply consists in a drawn widely panned *Spectrum* with a predefined transformation graph, containing alternative paths. As seen at the end of the previous section, the number keys on the computer keyboard can be used to directly select states in rapid succession. In this example I did this to add a rhythmic quality, and I also used the knobs on a MIDI keyboard to spontaneously modify the transformations. The result is a changing spectral texture, each moment of which is based on a similar sound structure. Figure 10.10 shows state 5 of the process, from where the two alternative transformation fork off.

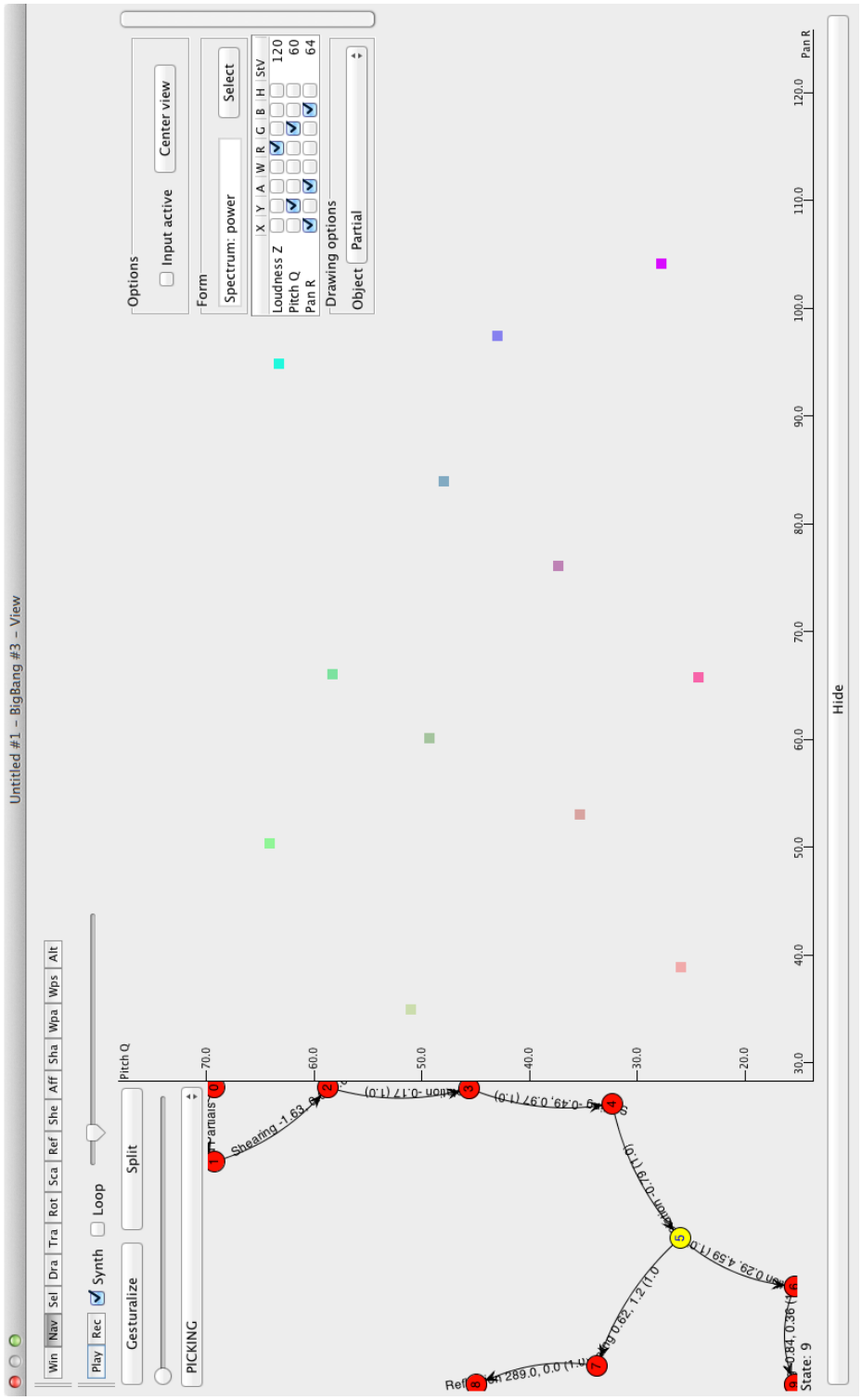


Figure 10.10: The *Spectrum* and process that form the basis of this brief improvisation.

10.2.2 Playing Sounds with a MIDI Keyboard and Modifying Them

Form *FMSet*

Graph a simple succession of transformations

Technique MIDI keyboard triggering and modifying

Output *BigBang* synth with different wave forms into Ableton Live

Link `http://www.soundcloud.com/bigbangrubette/designs`

The harmonic material in the previous example is inherent in the constellation of the *Spectrum* and its transformations. However, with *BigBang* it is also possible to regard its current contents as sonic material with which harmonic structures can be built by interaction with a MIDI keyboard, as described in Section 8.1.3. This example illustrates how *BigBang* can be used for sound design, by defining a few *FMSet* structures and modifying them. All sounds except the drum sounds were created in *BigBang* and played back using a MIDI keyboard, where each key plays a chromatic transposition of the current sound. Using the control change knobs of the keyboard, I modified the sounds while playing, which resulted in the various sweeping sounds in this example. An example of such an FM sound structure was shown in Figure 6.15.

10.2.3 Playing a MIDI Grand Piano with Leap Motion

Form *Spectrum*

Graph just an `AddObjectsOperation`

Technique drawing with Leap Motion

Output MIDI to a Steinway Grand Piano with the PianoDisc system

Link `http://youtu.be/ytGcKfhzF2Q`

In this example I used the Leap Motion controller newly made available to *BigBang*.[2] Specifically, I use the capability to add objects using Leap Motion in drawing mode, which quickly and dynamically adds and replaces the objects when the fingers move, described in detail in Section 8.1.3. I decided to use a *Spectrum* form played back with MIDI as quickly repeated notes instead of keeping keys pressed, which allows for fast rhythms and quick dynamic changes. In order to simulate the space of the piano keys – higher pitches to the right – and to have precise control over dynamics, I simply assigned *Pitch* to the x-axis view parameter, and *Loudness* to the y-axis, as shown in Figure 10.11. This results in a theremin-like dynamic setting, where the greater the distance of the hands from the Leap Motion controller, the louder the *Partials* of the *Spectrum*. The piece is fully improvised and starts out with monophonic melodic gestures played with one finger of the right hand, moving to a contrapuntal part with one finger of each hand. Later, I add more and more fingers to each hand, sometimes playing in parallel, sometimes independently, culminating in a part with increasingly energetic and fast gestures, at an increasing distance from the Leap Motion, culminating in a loud and choppy part of thrown gestures.

10.2.4 Playing a MIDI Grand Piano with the Ableton Push

Form *Spectrum*

Graph four sequential transformations

[2] Tormoen, Thalmann, and Mazzola, "The Composing Hand: Musical Creation with Leap Motion and the BigBang Rubette".

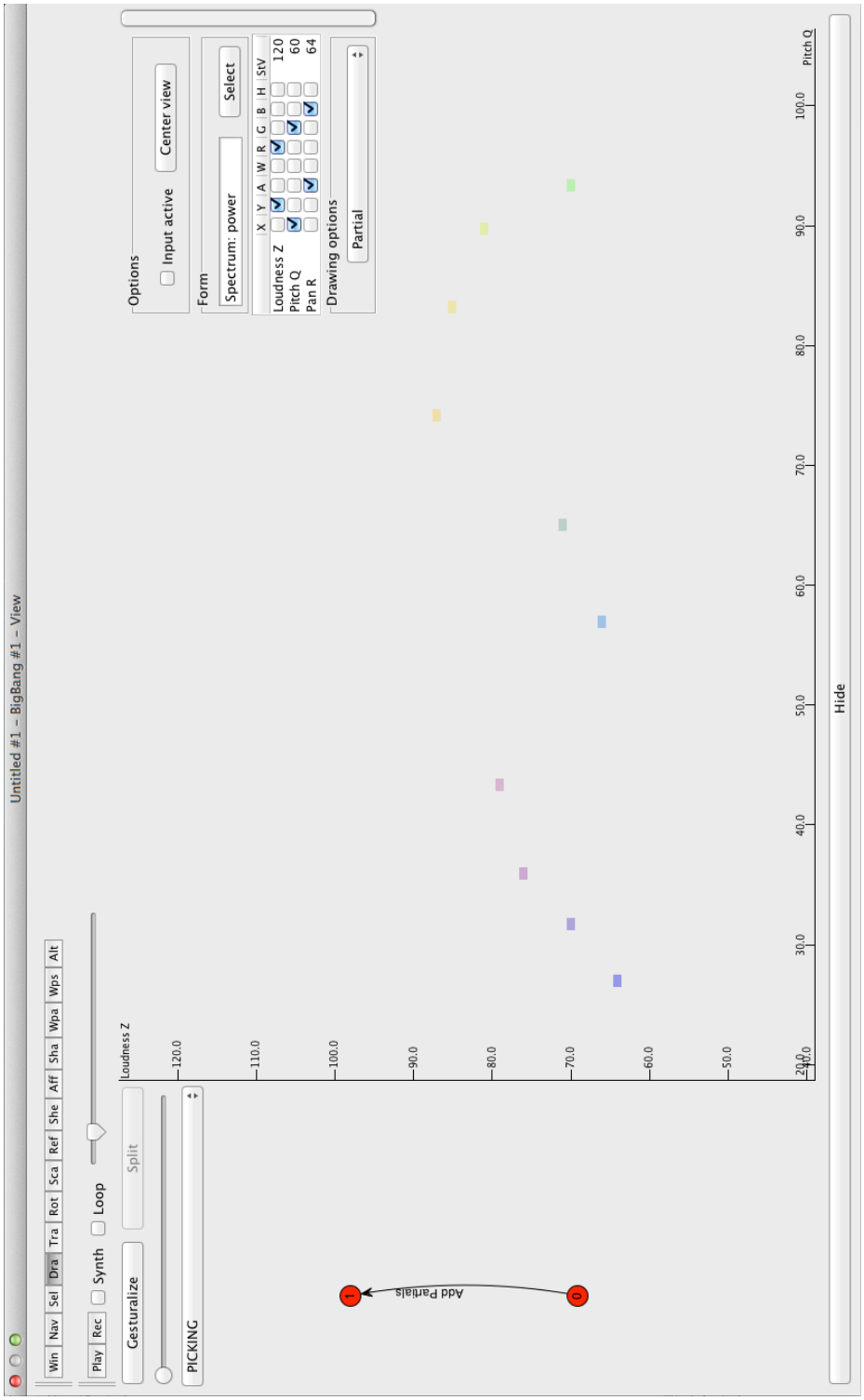


Figure 10.11: Two piano hands drawn with Leap Motion on the *Pitch* × *Loudness* plane.

Technique aftertouch dynamics, manual gesturalizing using the Ableton Push

Output MIDI to a Steinway Grand Piano with the PianoDisc system

Link `http://youtu.be/n2Pi281XZP4`

As opposed to the previous example, the simple setup of which enabled maximal freedom of determining the structure of the piece and its tonal material spontaneously, this example shows how it is also possible to improvise with some deliberately prepared material. The piece uses the same form, *Spectrum*, with the same rate of MIDI note repetition, but it also includes a simple graph that predefines intervallic and transformative gestural material. The transformed entity is a four-note motive of purely harmonic nature, due to the *Spectrum*'s atemporal quality. Figure 10.12 shows the initial motive and the graph. First, the motive is rotated by about 180 degrees, then scaled, mainly in pitch (about 4.5 times larger), then rotated counterclockwise by slightly less than 90 degrees, and finally scaled to an intervallic and dynamic unison.

After defining the simple graph, I decided to perform the piece using the dynamic capabilities of the Ableton Push controller. The Push offers pads sending note on/off messages, which I mapped to played back versions of the four-note motive, in a similar way the sounds were triggered by the MIDI keyboard in the example in Section 10.2.2. More importantly, the Push's pads send out highly precise monophonic aftertouch control changes, which I used to control the dynamics of the played back motives. Furthermore, I mapped the Push's large touch strip to the *BigBang*'s manual gesturalizing slider, which allowed me to freely move back and forth through the various transformed versions of the motive, both continuously by sliding, and discretely by tapping the strip, which jumps to the corresponding position of the gesturalization (this way users can not only select the composition states represented by

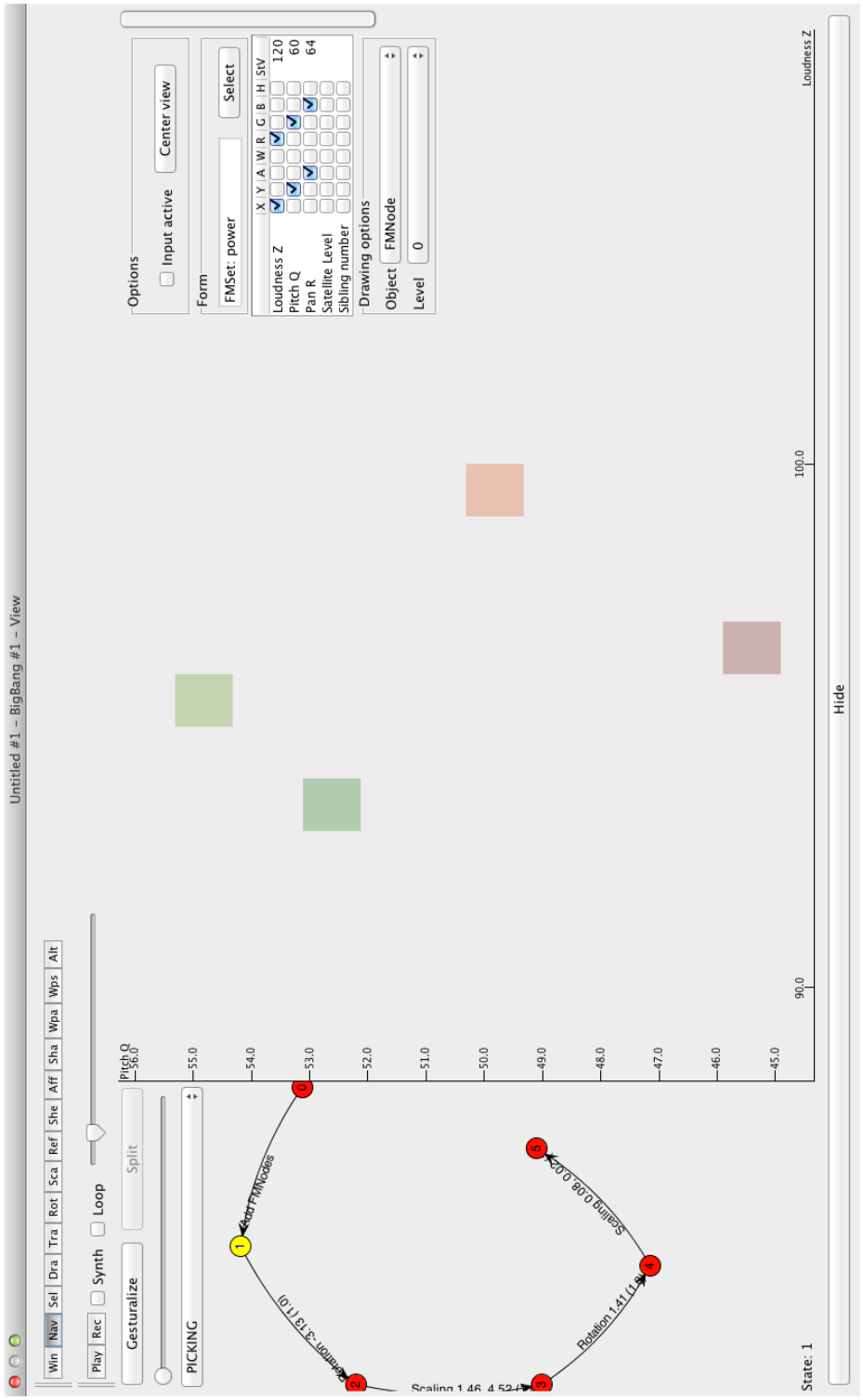


Figure 10.12: The initial motive and the simple sequential graph.

the nodes of the graph, but any other intermediary state!). In the beginning of the improvisation, I gradually add more pitches by gesturalizing the `AddObjectsOperation`, then I spontaneously move through different states of the motive's transformational path, reacting to the temporary constellations by playing them in different ways on the pads. In the end, the motive disappears in a unison produced by the ultimate scaling.

10.2.5 Improvising with 12-Tone Rows

Form *Score*

Graph a sequential transformation graph with a few identities

Technique gesturalizing and looping, aftertouch dynamics with the Ableton Push

Output MIDI to a Steinway with PianoDisc

Link `http://youtu.be/n1RQimytD2A`

This last example uses a similar setup as the previous one, playing the MIDI Grand with the Push. However, it is even more deliberately prepared, using automatic gesturalization, and links to the second musical example presented here (Section 10.1.2), in its use of the *Score* form in combination with looping while the graph is being gesturalized. Here, the base material is a simple twelve-tone row, generated by another rubette I recently created, the *NTone* rubette, which generates rows of N equidistant tones, microtonal if necessary, within a specific interval of I semitones (here, $N = 12$ and $I = 12$). The twelve-tone row is then transformed in simple ways (Figure 10.13 shows the original row and the graph). It is first compressed (scaled) in *Onset*, which results in a faster version, then expanded in *Onset*, leading to a staccato version. At this point, I inserted an identity scaling in order to achieve a static moment in

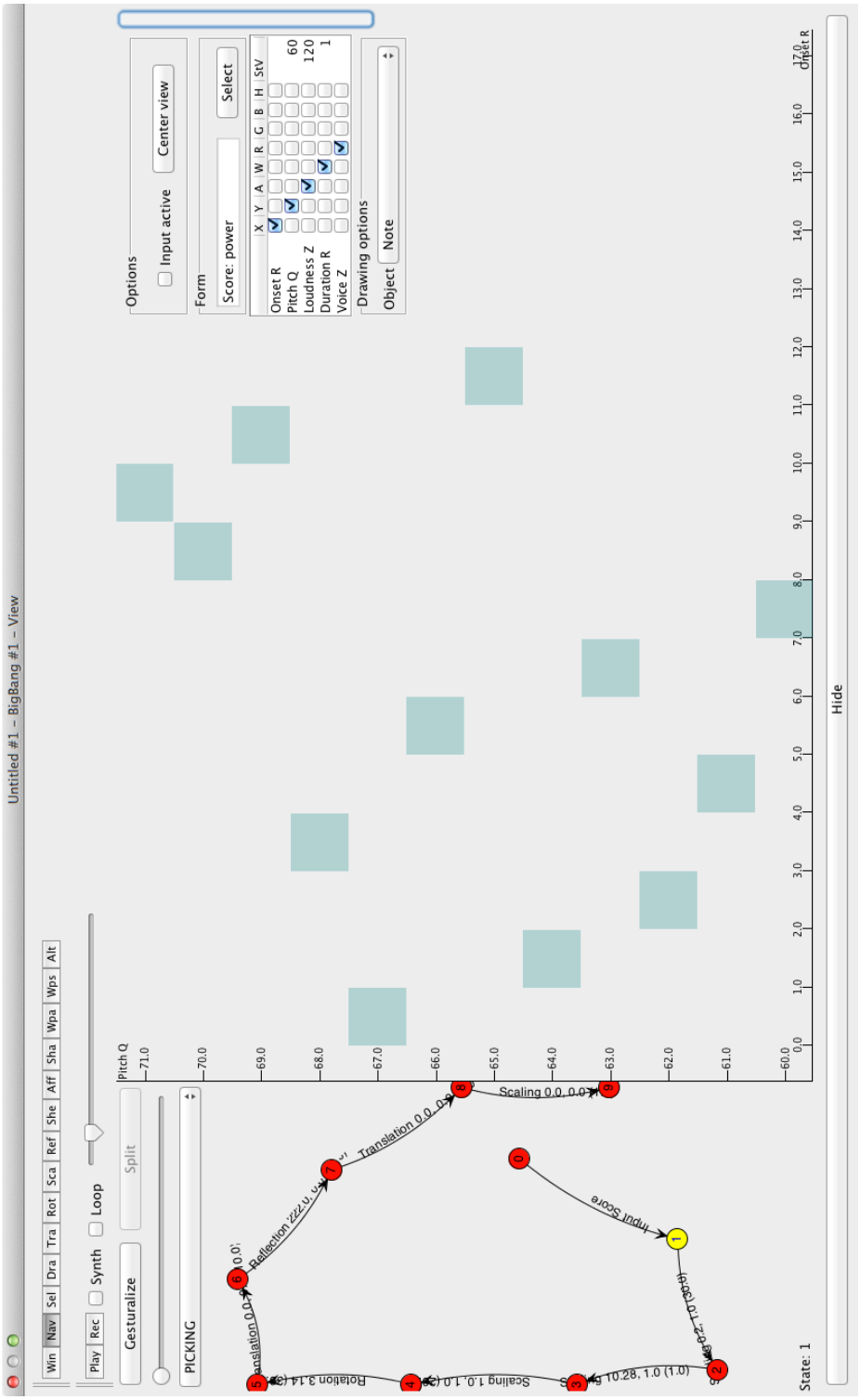


Figure 10.13: The sequential graph with the initial state selected, showing the original twelve-tone row.

gesturalization, as suggested in Section 10.1.6. After that, the row gradually assumes its retrograde inversion form by being rotated by 180 degrees, again stays static, is inverted by a reflection, again stays static, and finally disappears in a single pitch (scaling to 0 in both onset and pitch). In the performed version, again each pad of the Push triggers a transposed version of the twelve-tone row, again dynamically modulated by aftertouch. In this case, I could fully focus on playing the Push, since gesturalization was predetermined and happened automatically.

Bibliography

Adorno, Theodor W. *Zu einer Theorie der musikalischen Reproduktion*. Frankfurt am Main: Suhrkamp, 2001.

Alpaydin, Ruhan and Guerino Mazzola. "A Harmonic Analysis Network". In: (forthcoming).

Alperson, Philip. "On musical improvisation". In: *Journal of Aesthetics and Art Criticism* 43.1 (1984), pp. 17–29.

Ben-Tal, Oded. "Characterising musical gestures". In: *Musicae Scientiae* 16.3 (2012), pp. 247–61.

Bevilacqua, Frédéric and Remy Muller. "A gesture follower for performing arts". In: *Proceedings of the International Gesture Workshop*. 2005.

Boulez, Pierre. *Jalons*. Paris: Bourgeois, 1989.

— "Sonate, que me veux-tu?" In: *Perspectives of New Music* (1963), pp. 32–44.

Brown, Matthew and Douglas J. Dempster. "The Scientific Image of Music Theory". In: *Journal of Music Theory* 33.1 (1989), pp. 65–106.

Burk, Phil. "JSyn – A Real-time Synthesis API for Java". In: *Proceedings of the International Computer Music Conference*. San Francisco: International Computer Music Association, 1998.

Chadabe, Joel. "Interactive Composing: An Overview". In: *Computer Music Journal* 8.1 (1984), pp. 22–7.

Châtelet, Gilles. *Figuring space: philosophy, mathematics, and physics*. Kluwer, 2000.

Cherlin, Michael. "On Adapting Theoretical Models from the Work of David Lewin". In: *Indiana Theory Review* 14 (1993), pp. 19–43.

Chowning, John. "The Synthesis of Complex Audio Spectra by Means of Frequency Modulation". In: *Journal of the Audio Engineering Society* 21 (1973).

Clynes, Manfred. "Microstructural Musical Linguistics: composer's pulses are liked best by the best musicians". In: *COGNITION, International Journal of Cognitive Science* 55 (1995), pp. 269–310.

Collins, David. "A synthesis process model of creative thinking in music composition". In: *Psychology of Music* 33.2 (2005), pp. 193–216.

Cope, David. *Computer Models of Musical Creativity*. Cambridge, MA: MIT Press, 2005.

Coutaz, Joëlle. "PAC, an implementation model for dialog design". In: *Proceedings of INTERACT*. 1987, pp. 431–6.

Csíkszentmihályi, Mihály. *Flow: The Psychology of Optimal Experience*. New York: Harper and Row, 1990.

Davies, David. *Art as Performance*. Malden, MA: Blackwell, 2004.

Davies, Stephen. *Musical works and performances: A philosophical exploration*. Oxford University Press, 2001.

DeBellis, Mark. *Music and Conceptualization*. Cambridge University Press, 1995.

Forte, Allen. *The Structure of Atonal Music*. New Haven and London: Yale University Press, 1973.

Gabrielsson, Alf and Erik Lindström. "Emotional expression in synthesizer and sentograph performance". In: *Psychomusicology: Music, Mind & Brain* 14.1 (1995), pp. 94–116.

Geirland, John. “Go With The Flow”. In: *Wired magazine* 4.09 (Sept. 1996).

Gentilucci, Maurizio and Michael C Corballis. “From manual gesture to speech: A gradual transition”. In: *Neuroscience & Biobehavioral Reviews* 30.7 (2006), pp. 949–60.

Goehr, Lydia et al. “Philosophy of music”. In: *Grove Music Online. Oxford Music Online*. Oxford University Press, 2014.

Goeller, Stefan. “Object Oriented Rendering of Complex Abstract Data”. Ph.D. Thesis. Universität Zürich, 2004.

Goldin-Meadows, Susan. *Hearing Gesture: How Our Hands Help Us Think*. Harvard University Press, 2003.

Gould, Carol S and Kenneth Keaton. “The essential role of improvisation in musical performance”. In: *Journal of Aesthetics and Art Criticism* 58.2 (2000), pp. 143–8.

Graeser, Wolfgang. “Bachs Kunst der Fuge”. In: *Bach-Jahrbuch* (1924), p1ff.

Hanson, Howard. *Harmonic Materials of Modern Music: Resources of the Tempered Scale*. New York: Appleton-Century-Crofts, 1960.

Hatten, Robert. “A theory of musical gestures and its application to Beethoven and Schubert”. In: *Music and Gesture*. Ed. by A. Gritten and E. King. Aldershot: Ashgate, 2006.

— *Interpreting Musical Gestures, Topics, and Tropes*. Indiana University Press, 2004.

Hook, Julian. “David Lewin and the Complexity of the Beautiful”. In: *Intégral* 27 (2007), pp. 55–90.

— “Uniform Triadic Transformations”. In: *Journal of Music Theory* 46.1/2 (2002), pp. 57–126.

Hunt, Andy and Ross Kirk. “Mapping Strategies for Musical Performance”. In: *Trends in Gestural Control of Music*. Ed. by M.M. Wanderley and M. Battier. Paris: Ircam - Centre Pompidou, 2000.

Hunt, Andy and Marcelo M. Wanderley. “Mapping performer parameters to synthesis engines”. In: *Organised Sound* 7.2 (2002), pp. 97–108.

Kania, Andrew. “The Philosophy of Music”. In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Spring 2014 Edition. 2014.

Klein, Michael Leslie. *Intertextuality in Western art music*. Indiana University Press, 2005.

Klumpenhouwer, Henry. “Essay: In Order to Stay Asleep as Observers: The Nature and Origins of Anti-Cartesianism in Lewin’s Generalized Musical Intervals and Transformations”. In: *Music Theory Spectrum* 28:2 (2006), pp. 277–89.

Krasner, Glenn E. and Stephen T. Pope. “A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80”. In: *Journal of Object-Oriented Programming* 1.3 (1988), pp. 26–49.

Laufs, Uwe, Christopher Ruff, and Anette Weisbecker. “Mt4j: an open source platform for multi-touch software development”. In: *VIMation Journal* (2010).

Leman, Marc. *Embodied Music Cognition and Mediation Technology*. Cambridge: MIT Press, 2007.

— “Music, Gesture, and the Formation of Embodied Meaning”. In: *Musical gestures : sound, movement, and meaning*. Ed. by Rolf Inge Godøy and Marc Leman. New York: Routledge, 2010.

Levinson, Jerrold. *Music in the Moment*. Cornell University Press, 1997.

Levitin, Daniel J., Stephen McAdams, and Robert L. Adams. “Control parameters for musical instruments: a foundation for new mappings of gesture to sound”. In: *Organised Sound* 7.2 (2002), pp. 171–89.

Lewin, David. “A Formal Theory of Generalized Tonal Functions”. In: *Journal of Music Theory* 26:1 (1982), pp. 23–60.

— “Forte’s Interval Vector, My Interval Function, and Regener’s Common-Note Function”. In: *Journal of Music Theory* 21:2 (1977), pp. 194–237.

— *Generalized Musical Intervals and Transformations*. New York, NY: Oxford University Press, 1987/2007.

— “Klumpenhouwer networks and some isographies that involve them”. In: *Music Theory Spectrum* 12.1 (1990), pp. 83–120.

— “Music Theory, Phenomenology, and Modes of Perception”. In: *Music Perception* 3 (1986), pp. 327–92.

— *Musical Form and Transformation: Four Analytic Essays*. New Haven: Yale University Press, 1993.

Lohner, H. and I. Xenakis. “Interview with Iannis Xenakis”. In: *Computer Music Journal* 10.4 (1986), pp. 50–5.

Marino, G., M.-H. Serra, and J.-M. Raczinski. “The UPIC System: Origins and Innovations”. In: *Perspectives of New Music* 31.1 (1993), pp. 258–69.

Mazzola, Guerino. *Geometrie der Töne: Elemente der Mathematischen Musiktheorie*. Basel: Birkhäuser, 1990.

— *Gruppen und Kategorien in der Musik: Entwurf einer mathematischen Musiktheorie*. Berlin: Heldermann Verlag, 1985.

— *La vérité du beau dans la musique*. Paris: Delatour/IRCAM, 2007.

— *Musical Performance: A Comprehensive Approach: Theory, Analytical Tools, and Case Studies*. Berlin Heidelberg: Springer, 2011.

— *Synthesis*. Zürich: SToA music CD ST-71.1001, 1990.

— *The Topos of Music. Geometric Logic of Concept, Theory, and Performance*. Basel: Birkhäuser, 2002.

Mazzola, Guerino and Moreno Andreatta. “Formulas, Diagrams, and Gestures in Music”. In: *Journal of Mathematics and Music* 1.1 (2007), pp. 21–32.

— “From a Categorical Point of View: K-nets as Limit Denotators”. In: *Perspectives of New Music* 44.2 (2006).

Mazzola, Guerino and Paul Cherlin. *Flow, Gesture and Spaces in Free Jazz. Towards a Theory of Collaboration*. Berlin/Heidelberg: Springer, 2009.

Mazzola, Guerino, Joomi Park, and Florian Thalmann. *Musical Creativity – Strategies and Tools in Composition and Improvisation*. Heidelberg et al.: Springer Series Computational Music Science, 2011.

Mazzola, Guerino and Florian Thalmann. “Musical Composition and Gestural Diagrams”. In: *Proceedings of the Third International Conference on Mathematics and Computation in Music (MCM)*. Ed. by C. Agon et al. Heidelberg: Springer, 2011.

Mazzola, Guerino et al. “Functors for Music: The Rubato Composer System”. In: *Digital Art Weeks Proceedings*. Zürich: ETH, 2006.

Merleau-Ponty, Maurice. *Phénoménologie de la perception*. Gallimard, 1945.

Milmeister, Gérard. *The Rubato Composer Music Software: Component-Based Implementation of a Functorial Concept Architecture*. Berlin/Heidelberg: Springer, 2009.

Miranda, Eduardo Reck and Marcelo M Wanderley. *New digital musical instruments: control and interaction beyond the keyboard*. Vol. 21. Computer music and digital audio series. Middleton: A-R Editions, 2006.

Molino, Jean. “Musical Fact and the Semiology of Music”. In: *Music Analysis* 9.2 (1990), pp. 105–56.

Montiel Hernandez, Mariana. “El Denotador: Su Estructura, construcción y Papel en la Teoría Matemática de la Musica”. MA thesis. Mexico City: UNAM, 1999.

Morris, Robert D. *Composition with pitch-classes: a theory of compositional design*. New Haven: Yale University Press, 1987.

Mulder, Axel G. E. "Towards a choice of gestural constraints for instrumental performers". In: *Trends in Gestural Control of Music*. Ed. by M. M. Wanderley and M. Battier. Ircam - Centre Pompidou, 2000.

Nattiez, Jean-Jacques. *Musicologie générale et sémiologue*. Paris: Christian Bourgois, 1987.

Nienhuys, Han-Wen and Jan Nieuwenhuizen. "LilyPond, a system for automated music engraving". In: *Proceedings of the XIV Colloquium on Musical Informatics*. Firenze: CIM, 2003.

O'Madadhain, Joshua et al. *The JUNG (Java Universal Network/Graph) Framework*. URL: `http://jung.sourceforge.net/index.html`.

Perle, George. "Pitch-class set analysis: An evaluation". In: *Journal of Musicology* (1990), pp. 151–72.

Puckette, Miller. "Pure Data: another integrated computer music environment". In: *Proceedings of the International Computer Music Conference*. 1996, pp. 37–41.

Puckette, Miller and David Zicarelli. "MAX - An interactive graphic programming environment". In: *Opcode Systems, Menlo Park, CA* (1990).

Rahn, John. "Cool tools: Polysemic and non-commutative Nets, subchain decompositions and cross-projecting pre-orders, object-graphs, chain-hom-sets and chain-label-hom-sets, forgetful functors, free categories of a Net, and ghosts". In: *Journal of Mathematics and Music* 1.1 (2007), pp. 7–22.

Rings, Steven. "Tonality and Transformation". Ph.D. Thesis. Yale University, 2006.

Roeder, John. "Constructing Transformational Signification: Gesture and Agency in Bartok's Scherzo, Op. 14, No. 2, measures 1-32". In: *Music Theory Online* 15:1 (2009).

Satyendra, Ramon. “An Informal Introduction to Some Formal Concepts from Lewin’s Transformational Theory”. In: *Journal of Music Theory* 48 (2004), pp. 99–141.

Sawyer, R. Keith. *Group Creativity: Music, Theater, Collaboration*. Mahwah, NJ: Lawrence Erlbaum Associates, Publishers, 2003.

Scruton, Roger. *The Aesthetics of Music*. Oxford: Oxford University Press, 1997.

Seibt, Johanna. “Process Philosophy”. In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Fall 2013. 2013.

Selfridge-Field, Eleanor. *Beyond MIDI: the handbook of musical codes*. Cambridge, MA: MIT Press, 1997.

Shepard, Roger N and Lynn A Cooper. “Mental Images and Their Transformations”. In: *MIT Press*. 1986.

Taruskin, Richard. “Review of Forte, The Harmonic Structure of the Rite of Spring”. In: *Current Musicology* 28 (1979), p. 119.

Thalmann, Florian. “Musical composition with Grid Diagrams of Transformations”. Master’s Thesis. University of Bern, 2007.

Thalmann, Florian and Guerino Mazzola. “Affine Musical Transformations Using Multi-touch Gestures”. In: *Ninad* 24 (2010), pp. 58–69.

— “Gestural Shaping and Transformation in a Universal Space of Structure and Sound”. In: *Proceedings of the International Computer Music Conference*. New York City: International Computer Music Association, 2010.

— “Poietical Music Scores: Facts, Processes, and Gestures”. In: *Proceedings of the Second International Symposium on Music and Sonic Art*. Baden-Baden: MuSA, 2011.

— "The BigBang Rubette: Gestural Music Composition with Rubato Composer". In: *Proceedings of the International Computer Music Conference*. Belfast: International Computer Music Association, 2008.

Thalmann, Florian and Guerino Mazzola. "Using the Creative Process for Sound Design based on Generic Sound Forms". In: *MUME 2013 proceedings*. Boston: AAAI Press, 2013.

— "Visualization and Transformation in a General Musical and Music-Theoretical Spaces". In: *Proceedings of the Music Encoding Conference 2013*. Mainz: MEI, 2013.

Tormoen, Daniel, Florian Thalmann, and Guerino Mazzola. "The Composing Hand: Musical Creation with Leap Motion and the BigBang Rubette". In: *Proceedings of 14th International Conference on New Interfaces for Musical Expression (NIME)*. London, 2014.

Tymoczko, Dmitri. *A Geometry of Music*. New York: Oxford University Press, 2011.

Wanderley, Marcelo M. "Gestural Control of Music". In: *International Workshop Human Supervision and Control in Engineering and Music*. 2001, pp. 632–644.

Wessel, David and Matthew Wright. "Problems and Prospects for Intimate Musical Control of Computers". In: *Computer Music Journal* 26.3 (2002), pp. 11–22.

Wilkie, Katie, Simon Holland, and Paul Mulholland. "What Can the Language of Musicians Tell Us about Music Interaction Design?" In: *Computer Music Journal* 34.4 (2010), pp. 34–49.

Wittlich, Gary. "Sets and Ordering Procedures in Twentieth-Century Music". In: *Aspects of Twentieth-Century Music*. Ed. by Gary Wittlich. Englewood Cliffs, New Jersey: Prentice-Hall, 1975.

Wu, Dan et al. "Music composition from the brain signal: representing the mental state by music". In: *Computational intelligence and neuroscience* (2010).

Xenakis, Iannis. *Musiques Formelles*. Paris: Editions Richard-Masse, 1963.

Young, James O. and Carl Matheson. "The metaphysics of jazz". In: *Journal of Aesthetics and Art Criticism* 58.2 (2000), pp. 125–33.

Zamborlin, Bruno et al. "Fluid Gesture Interaction Design: Applications of Continuous Recognition for the Design of Modern Gestural Interfaces". In: *ACM Transactions on Interactive Intelligent Systems* 3.4 (2014).

Zbikowski, Lawrence M. *Conceptualizing music: Cognitive structure, theory, and analysis*. Oxford University Press, 2002.

Zuckerkandl, Victor. *Sound and Symbol. Music and the External World*. Ed. by Translated by Willard R. Trask. Routledge, 1956.