GRANCLOUD: A REAL-TIME GRANULAR SYNTHESIS APPLICATION AND ITS IMPLEMENTATION IN THE INTERACTIVE COMPOSITION *CREO*

Terry Alan Lee, B.M.

Thesis Prepared for the Degree of

MASTER OF MUSIC

UNIVERSITY OF NORTH TEXAS

December 2009

APPROVED:

Cindy McTee, Major Professor
Jon Christopher Nelson, Minor Professor
Andrew May, Committee Member
Joseph Klein, Chair, Division of Composition Studies
Graham H. Phipps, Director of Graduate Studies
James C. Scott, Dean of the College of Music
Michael Monticino, Dean of the Robert B. Toulouse School of Graduate Studies

Lee, Terry Alan. GranCloud: A real-time granular synthesis application and its implementation in the interactive composition *Creo*. Master of Music (Composition), December 2009, 127 pp., 9 tables, 22 illustrations, references, 18 titles.

GranCloud is new application for the generation of real-time granular synthesis in the SuperCollider programming environment. Although the software was initially programmed for use in the interactive composition *Creo,* it was implemented as an independent program for use in any computer music project. GranCloud consists of a set of SuperCollider classes representing granular clouds and parameter objects defining control data for the synthesis. The software is very flexible, allowing users to create their own grain synthesis definitions and control parameters. Cloud objects encapsulate all of the control data and methods necessary to render virtually any type of granular synthesis. Parameter objects provide several simple methods for mapping grain parameters to complex changing data sets or to external data sources. GranCloud simplifies the complex task of generating granular synthesis, allowing composers to focus less on technological issues and more on musical considerations during the compositional process.

TABLE OF CONTENTS

Page

LIST OF TABLES iv

LIST OF FIGURES v

PART I: CRITICAL ANALYSIS OF GRANCLOUD 1

1. Introduction 2

2. GranCloud: Design and Purpose 17

3. GranCloud: Class Structure 44

4. Application of GranCloud in *Creo* 76

5. Conclusion 92

Reference List 93

PART II: *CREO* SCORE 95

Title Page 96

Program Notes 97

Performance Notes 98

Score 99

LIST OF TABLES

Table 1. GranCloud Instance Variables Used for the Control Data Repository...26

Table 2. Parameter Object Classes Included with GranCloud 30

Table 3. Cloud Class Inheritance and Descriptions 45

Table 4. Instance Variables of the GranCloudSimple Class 46

Table 5. Methods and Parameter Descriptions of the GranCloudSimple Class..50

Table 6. Instance Variables of the GranCloud2 Class 53

Table 7. Methods and Parameter Descriptions of the GranCloud2 Class.....54

Table 8. Parameter Class Inheritance and Descriptions.....56

Table 9. Methods Common to all Parameter Object Classes 59

LIST OF FIGURES

Fig. 1. Examples of grain waveforms: a) sine wave, b) audio sample.....6

Fig. 2. Examples of grain envelope shapes: a) quasi-gaussian, ...7

Fig. 3. The SuperCollider client-server architecture 11

Fig. 4. Typical SuperCollider client-server application logic 14

Fig. 5. GranCloud architectural design.....21

Fig. 6. Code example showing the creation of a SynthDef named *sine_grain* 24

Fig. 7. Grain waveforms generated using the sine_grain SynthDef 25

Fig. 8. Parameter object access and mutation using the *at* and *put* methods.....33

Fig. 9. GranCloud synthesis scheduling engine flowchart.....36

Fig. 10. GranCloud *prepareGrain* method flowchart 39

Fig. 11. Code examples of GCValue object instantiation 62

Fig. 12. Code examples of the instantiation of GCMinMax 64

Fig. 13. Code examples of the instantiation GCCenterDev objects 66

Fig. 14. Code example of a GCControlMap 68

Fig. 15. Code example of instantiating a GCTrajectoryMap.....70

Fig. 16. Code example of the instantiation of a GCBusMap object 71

Fig. 17. Code example of the instantiation of a GCMidiMap object.....72

Fig. 18. Example of harmonization in *Creo* using GranCloud 77

Fig. 19. Sine grain SynthDef used in *Creo* for low rumble 82

Fig. 20. Blip grain SynthDef used in *Creo* for tinkling sound 83

Fig. 21. Buffer grain SynthDef used in *Creo* for breaking glass sounds.....85

Fig. 22. GranCloud code example of water sound transformation 86

PART I

CRITICAL ANALYSIS OF GRANCLOUD

AND ITS APPLICATION TO *CREO*

Chapter 1

Introduction

GranCloud and Creo

GranCloud is a new application for the generation of real-time granular synthesis in the SuperCollider[1] programming environment. It was designed to be highly flexible, extendable, and efficient, so it could generate virtually any type of granular synthesis. I programmed the software in response to several specific needs encountered during the composition of *Creo,* an interactive work for a small group of chamber instruments and computer.

Creo was written for flute, violin, French horn, piano, and live interactive electronics. Thematic elements of the composition are closely related to the interpretation of its Latin title—meaning, "to create, make, or produce."[2] The composition uses the transformation of sonic textures to metaphorically portray scientific and religious theories related to creation.

Many of these changing textures are created in the computer music using granular synthesis. By changing various parameters of the synthesis over time,

---

[1] SuperCollider is an open source synthesis environment and programming language for real-time audio synthesis and algorithmic composition. More information about the environment can be found at the SuperCollider project website. "SuperCollider: real-time audio synthesis and algorithmic composition," [resource on-line]; available from http://supercollider.sourceforge.net; Internet; accessed 1 October 2009.

[2] *The Pocket Oxford Latin Dictionary (Latin-English),* 1994 ed., s.v. "creo."

chaotic clouds of sound are morphed into ordered textures and recognizable sounds. These subtle transformations require the use of a highly flexible granular synthesis application.

The core apparatus used by *Creo* is a computer running a SuperCollider program called a controller script that governs all aspects of the computer music. The controller analyzes microphone signals from each instrument to track the progression of the performance through the score and to cue the generation of computer music events at appropriate times. The controller interacts directly with the granular synthesis application to define, generate, and control granular clouds used in the composition.

Since the progression of time through the composition is dependent on human interpretation which may vary in different performances, the computer music must be able to adjust elements of the synthesis in real-time to ensure proper coordination. This added a real-time requirement to the granular synthesis application. It needed to be efficient enough to generate sound in a live setting and flexible enough to adapt immediately to live performance cues.

Creo utilizes custom-built spatialization software that defines localization parameters as trajectories on a three-dimensional spatial grid. The software pans signals to different channels based on these trajectories and a configurable object representing the speaker layout. The software allows the composition to be rendered properly on different speaker configurations by changing only a simple configuration object. The use of the software demanded a granular

synthesis application extendable enough to use the panning objects internally and to map panning parameters to the defined trajectories.

Generating granular synthesis in any setting can be a tedious and cumbersome task. There is a tremendous amount of control data to manage. The work involved in tracking the data, making grain specific calculations, scheduling grains, and setting grain parameters can slow down the compositional process significantly. Duplication of code can also become an issue as common code is copied and pasted from application to application in order to perform similar tasks. Considering these management difficulties along with the requirements discussed above, it became very clear to me that I would need a very powerful and flexible application for granular synthesis. Otherwise, the composition code would quickly become unmanageable.

Many good applications and tools have been developed to simplify the process of generating granular synthesis. Each has its own approach, strengths, and limitations. Most are focused on one or two specific types of granular synthesis. Many handle those synthesis methods very well, but I found none that fully met my requirements for *Creo*. Therefore, I decided to program my own application, GranCloud.

GranCloud allows users to define and render virtually any paradigm of granular synthesis within the limitations of the hardware being used. Grain synthesis definitions and control parameters are fully user-definable to maximize flexibility. All control data and methods needed to render granular clouds are

encapsulated within instances of GranCloud objects that simplify both the definition and generation of granular synthesis. The application abstracts the repetitive generational tasks from the composition code, allowing users to focus on more important compositional details.

Granular Synthesis

A basic understanding of granular synthesis is important to fully appreciate the capabilities of GranCloud and how it is useful to compositions such as *Creo*. Granular synthesis is based on the theory that any sound can be broken down into miniscule sonic events, that when added together, reproduce the whole[3]. Using granular synthesis, composers organize and place numerous tiny bits of sound in time and space to generate a complex composite sound.

Typically these grains of sound are extremely short in duration, often measured in milliseconds. They may occur at rates of thousands per second. A group of several grains organized in time sequentially form a composite sound that is often referred to as a cloud. Depending on the application, the grains may be organized synchronously, according to a specified pattern, or asynchronously with respect to time and space. They may be arranged in succession, or they may overlap. Changing the shape, duration, sonic content, delivery rate, synchronicity, and spatial position of the individual grains changes characteristics of the resultant sound. The possible sonic variations are nearly endless.

---

[3] Curtis Roads, *Microsound.* (Cambridge, Massachusetts: The MIT Press, 2001), 57-58.

The Composition of a Grain

In practice, any small bit of sound may function as a grain. When speaking of granular synthesis, however, most composers define a grain as a fragment of a waveform shaped by an amplitude envelope[4]. Figure 1 shows two examples of grains waveforms. The first is based on a sine wave enveloped within a quasi-Gaussian envelope. The second is a fragment of an audio sample enveloped in the same manner.

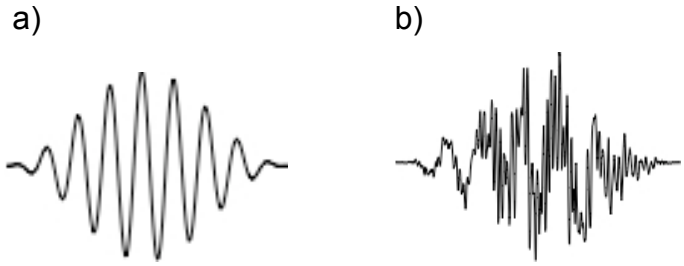


Fig. 1. Examples of grain waveforms: a) sine wave, b) audio sample.

Grain waveforms may be derived from any audio source. They may be synthetically generated, taken from a buffered audio sample, or taken from a live audio signal. A new signal may be synthesized specifically for each grain, or the grains may use fragments of an external signal.

The waveform is typically windowed within an amplitude envelope. The envelope provides shaping and suppresses clicks and high frequency noise

[4] Ibid., 86-87.

generated by waveform beginnings and endings that do not occur at zero-crossings. Many applications that generate granular synthesis use only a single envelope shape, typically quasi-Gaussian, that may not be altered from grain to grain. Composers often, however, prefer to utilize more than one envelope shape, or to transform the shape over time, in order to alter sonic characteristics of the cloud. Figure 2 shows several examples of grain envelope shapes commonly used in granular synthesis applications.

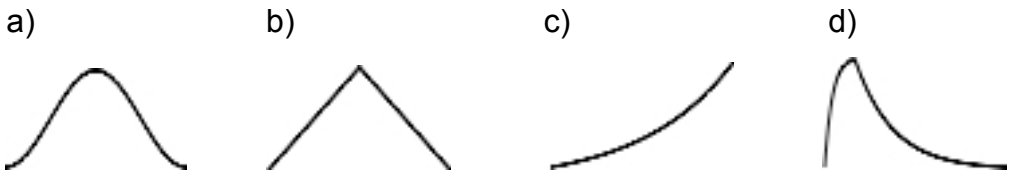


Fig. 2. Examples of grain envelope shapes: a) quasi-gaussian, b) triangle, c) exponential, d) percussive.

Effects processing is often performed on or within grains while generating granular synthesis. Processing may be applied to the source waveform, the enveloped grain, or the cloud as a whole. While grain-by-grain effects processing can be used to create some interesting sounds, it comes at a high cost in terms of CPU cycles, so most composers apply effects to the cloud as a whole.

Clouds and Streams

Several individual grains sounding successively form a composite sound often referred to as a cloud or a stream. Sequences of grains organized in time stochastically are typically referred to as clouds, while synchronous sequences

may be termed either clouds or streams. Frequently, composers create textures of even greater complexity by overlapping multiple clouds.

The sonic character of a cloud is dependent on how individual grains are organized in time and space and on how the internal characteristics of grains change over time. These characteristics are usually controlled through the use of synthesis parameters that can receive different values for each grain. By changing the values from grain to grain, composers can shape the clouds into complex shifting textures or gestures. Typical grain parameters include grain rate or density, grain duration, waveform frequency or buffer read-rate, envelope shape, envelope amplitude, and pan position. It is also very common, however, to specify additional parameters to control various aspects of the synthesis or effects processing on the grain.

Control Data

It takes an enormous amount of control data to specify discrete values for every parameter of every grain. This is especially true in clouds with grain densities measured in hundreds or thousands of grains per second. Since it is usually impractical to explicitly define parameter values, they are typically generated algorithmically. Control data is then required only to define and control parameters of the algorithm.

The values generated may remain static from grain-to-grain, change continuously over time, or follow a specified pattern or distribution. It may be desirable for pattern or range of the distribution to change over time. Sometimes

grain parameters are dependent on the values of other parameters. These control methods can all be handled algorithmically, but allowing the implementation of all of them requires a highly flexibly granular application.

A Diversity of Granular Forms

There are many forms of granular synthesis with different characteristics, purposes, and implementations. Examples include matrices and screens on the time-frequency plane, pitch-synchronous overlapping streams, synchronous and quasi-synchronous streams, asynchronous clouds, physical or abstract models, and granulation of sampled sound[5]. Granular synthesis may be used to generate textures and gestures, change pitch or duration of a sampled sound, perform effects processing, for decorrelation and spatialization effects, and various of other purposes.

GranCloud is intended to be generic enough to handle any form of granular synthesis. This requires the user to do a little extra work to define the grain structure and the parameters used, but the trade off in flexibility and ease of use is typically well worth the added effort. Since *Creo* uses many different forms of granular synthesis, the flexibility inherent in GranCloud was an important feature of the design.

[5] Ibid., 92-98.

SuperCollider 3

A general knowledge of the architecture of the SuperCollider programming environment is important to fully understand the technical details discussed in later sections of this document. SuperCollider is both a programming language and software environment for the generation of real-time audio synthesis and algorithmic composition.[6] SuperCollider version 3 utilizes a client-server architecture consisting of two applications that work together to interpret control data and generate audio signals. Figure 3 shows a graphical representation of the SuperCollider architecture.

The Server Application

All audio processing is performed on the SuperCollider server application. Synthesis nodes are placed on an ordered execution tree on the server in response to messages from a client application. The server performs the calculations defined on the tree in order to generate and process audio signals. A synthesis node consists of a group of unit generators[7] that have been chained together and compiled into a reusable graph[8] structure. When a node is needed,

---

[6] "SuperCollider: real-time audio synthesis and algorithmic composition," [resource on-line]; available from http://supercollider.sourceforge.net; Internet; accessed 1 October 2009.

[7] Unit generators are highly optimized algorithms for producing audio or control signals on the SuperCollider server.

[8] A graph is a computer-science term for an abstract data structure consisting of a set of ordered node pairs. Compiling groups of calculation instructions into graph structures optimizes the initialization of nodes improving efficiency of the application.

the pre-compiled graph is duplicated and placed on the tree to perform the needed calculations in the proper order-of-execution[9] defined by the client application.

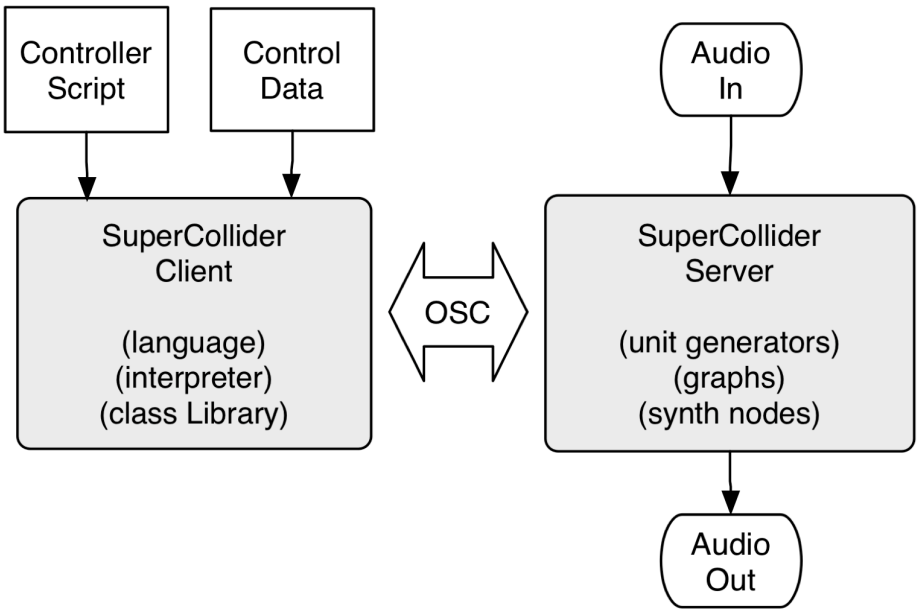


Fig. 3. The SuperCollider client-server architecture.

The server receives several types of instructions from the client. First, it receives SynthDef creation messages instructing the server how to connect unit generators for a specific type of synthesis node. Then, it receives node creation

[9] Order-of-execution refers the order in which synthesis node calculations are performed. If the inputs of a synthesis node are dependant on the output of other synthesis nodes then the controlling nodes must be placed on the execution tree before the dependant node.

messages via OSC[10] messaging. These messages include initial parameter values to use for the synthesis node by parameter name. The server also receives various types of control messages from the client via OSC. These messages control how synthesis nodes are chained or ordered and how values on busses should be routed and mapped to node parameters.

The Client Application

The SuperCollider client application consists of a programming language and interpreter. Composers write text-based computer code, called controller scripts, using the programming language. Then, they execute the scripts using the language interpreter. The interpreter reads the script, compiles it into machine language, and executes the machine code to generate the server messages at the appropriate times to control synthesis events.

The client programming language is a modern object-oriented[11] computer language closely related to Smalltalk with syntax similar to the C family of languages.[12] It has an extensive class library with various objects representing unit generators, synthesis nodes, busses, buffers, and other objects that are

---

[10] Open Sound Control is a standard messaging format for the transfer of audio and synthesis related information between software and electronic devices over a computer network.

[11] I use the term object-oriented in a computer science context, meaning a programming paradigm focused on data structures that include both data fields and methods. This is not to be confused with the process of graphically connecting objects with virtual patch cords as in Max/MSP or Pure Data.

[12] "SuperCollider," [resource on-line]; available from http://supercollider.sourceforge.net; Internet; accessed 1 October 2009.

used on the server. The class library also defines many object types representing different kinds of data structures to assist with the management and generation of control data as well as the timing and scheduling of events.

One of the strengths of the SuperCollider environment is that the client class library is easily extendable. Users may create and add their own classes to the library to expand the language and customize it to their own needs and desires. Users can extend the library simply by placing class definition files in the appropriate application directory.

Client-server Application Logic

Application logic for client-server interaction typically follows the standard pattern outlined in figure 4. At the top of the controller script, programmers define a Server object that represents the SuperCollider server application. The Server object contains the network address of the server so the client knows

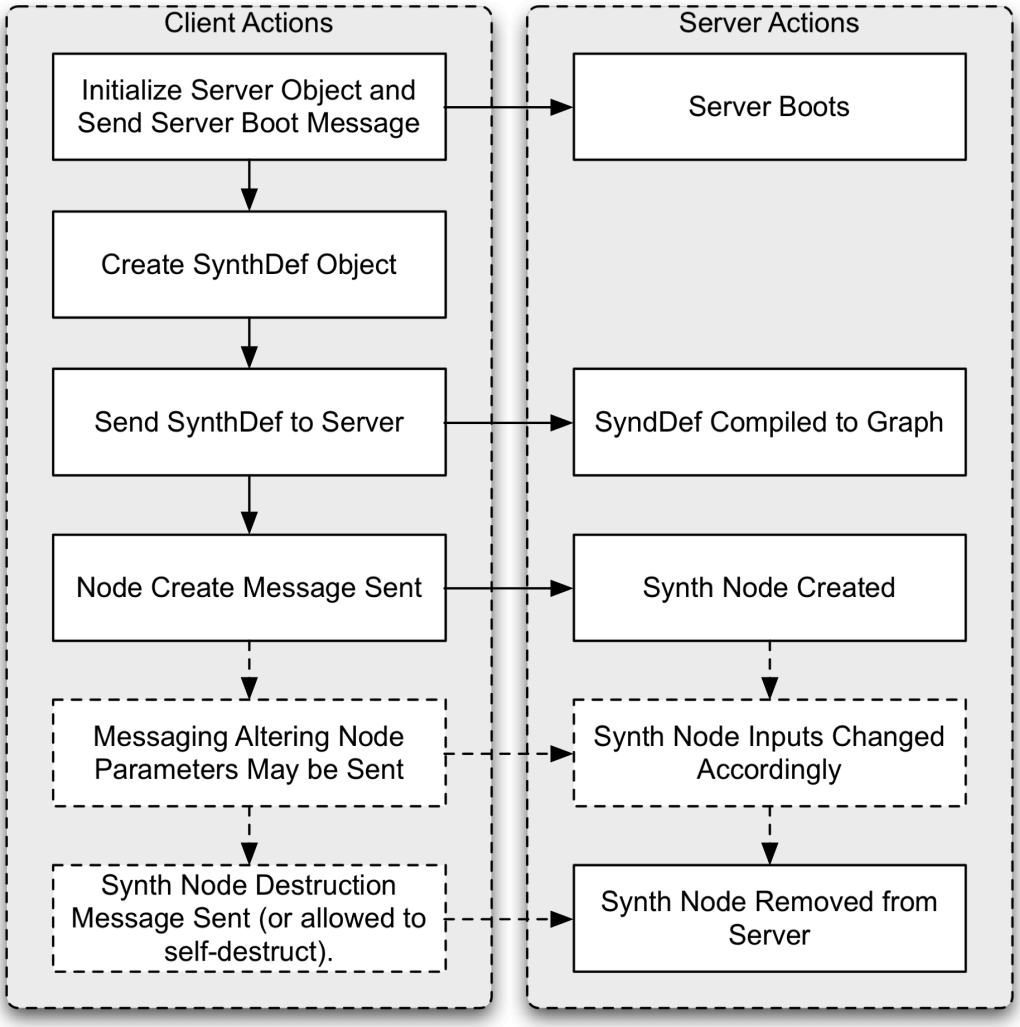


Fig. 4. Typical SuperCollider client-server application logic.

where to send messages. This object can be instructed to boot a server application at this address if it has not already been initialized.

Next, the controller script instantiates a set of SynthDef objects that act as templates for synthesis nodes that will be executed on the server. SynthDef objects contain a unique name and a unit generator graph function that defines how the synthesis should be performed. The function defines a set of input

parameters for the synthesis node as well as a description of how unit generators will be connected together to generate audio or control signals. The parameters allow composers to alter values used in the synthesis calculations.

Following the instantiation of SynthDef objects, the client application sends them to the server. The server compiles them into graphs that are retained in memory. Once they are compiled, they are ready to be used by the server as templates to create synthesis nodes when instructed by the client.

When it is time to execute a synthesis node for audio generation, the client sends a node creation message to the server. This message contains an execution time, the SynthDef name, parameters controlling order-of-execution, and a group of initial parameter values for the synthesis node. The client can also send messages to set values on control busses and map control busses to parameter inputs. When messages are received on the server, they are placed in a queue and executed at the scheduled time. The SuperCollider language includes a number of different objects to simplify the generation of the various types of server messages.

Once a synthesis node is no longer needed for signal generation it can be removed from the server either by an explicit message from the client or by an internal unit generator capable of removing the node once a trigger indicating node completion has been detected.

SuperCollider Applied in Creo

Although there are several synthesis frameworks suitable for interactive music, I chose SuperCollider for the composition of *Creo* for several reasons. It is a very efficient environment well suited for real-time audio applications. It is easily expandable, allowing users to program and incorporate their own tools into the environment itself. As an experienced programmer, I preferred text-based programming environments to graphical ones and object-oriented languages to procedural languages.

SuperCollider inherently provided all the basic building blocks I needed to handle the various digital signal analysis and sound synthesis tasks required by *Creo;* however, as I began developing the actual controller script for the composition, I soon realized that while these building blocks were very powerful, they were also very basic. I found myself programming a large amount of repetitive code to perform many common but complex tasks, such as the generation of granular synthesis. I recognized the need for additional tools to improve the manageability of my compositional process.

Chapter 2

GranCloud: Design and Purpose

GranCloud is a group of interrelated SuperCollider classes that I developed to simplify the process of generating granular synthesis. The present version of GranCloud is actually a second-generation rewrite of a previous project by the same name. Both versions share the same original concept, purpose, and design goals. I developed the current version as part of the *Creo* project in order to improve processing performance, add support for additional types of control objects needed by *Creo,* and to improve the flexibility and ease of use of the software.

The core of the GranCloud project is a cloud class named GranCloudSimple. It contains the granular synthesis engine and stores a group of helper objects used to manage grain parameter control data. When a GranCloudSimple object is properly instantiated, populated with parameter objects, and paired with a grain SynthDef object, it contains everything necessary to control the generation of a granular cloud on the SuperCollider server. The parameter objects provide several different methods of defining and controlling how grain parameter values change over time. Once plugged into a cloud object, the synthesis engine uses them to calculate the actual parameter values for each grain node executed on the server.

A secondary layer of the project includes the GranCloud2[13], GranCloudGroup, and GranCloud2Interface classes. The GranCloud2 class is a direct subclass of GranCloudSimple cloud class. It extends the functionality of the parent class to support several features, including the naming of cloud objects, the ability for multiple cloud objects to interact synchronously, the ability to save and retrieve cloud objects to and from external text files for reuse, and support features for graphical interfaces. The GranCloudGroup object allows multiple cloud instances to be grouped together and treated as a singular object for the execution of playback commands. The GranCloud2Interface class provides a graphical interface to build cloud objects and simplify the input and management of control data within them.

GranCloud Design

I designed GranCloud to be both simple to use and flexible enough to handle virtually any type of granular synthesis. Since there are many different granular synthesis paradigms and many ways that grains may be structured and organized in time and space, simple methods to maximize flexibility and extendibility were important to the design.

Early in the project I compiled a list of design requirements for the finished project. These requirements defined development goals and shaped structural

[13] The class was named GranCloud2 instead of GranCloud to maintain backwards compatibility with the first version of the software. This saves users who have compositions utilizing the first version from needing to convert the composition code in order to install and use the new version for other compositions.

decisions related to the project as a whole. Accomplishing these objectives required careful design and planning to ensure the final product would be fully functional and remain easy to use.

The design requirements included:

1. The finished product should be able to generate virtually any type of granular synthesis in real-time within the processing limitations of the hardware.

2. Clouds should be able to morph seamlessly from one state to another, even changing synthesis paradigms in the middle of the process.

3. Any sound source should be able to be used as the base sound for the granulation, including live audio signals, samples, or synthesized sound.

4. The grain envelope shape should be completely customizable and should be modifiable on a grain-by-grain basis.

5. Any type of signal processing should be performable on individual grains, and parameters controlling that processing should potentially be variable from grain to grain.

6. Redundant code for grain parameter calculation and grain scheduling should be abstracted by placing it within an internal synthesis engine.

7. All node creation, node destruction, and bus routing required by the synthesis should be handled within the synthesis engine.

8. Grain synthesis templates and parameter names should be fully customizable.

9. The synthesis engine and all control data for grain parameters should be encapsulated within a single object.

10. Users should be able to add as many grain parameters as their synthesis model requires.

11. Parameter values should be able to be interrelated, using the values of one or more parameters in the calculation of others.

12. Several methods of specifying and managing control data for grain parameters should be provided in order to use several different calculation paradigms for the generation of control data.

These design objectives required an application that is very customizable. This high level of design flexibility can become problematic within an application development process, because the programmer must anticipate the desires of users and provide methods for satisfying all possible needs. Through careful analysis and practical experimentation I was able to separate features requiring customization from routine tasks common to all types of granular synthesis. This allowed certain features to be left completely open-ended while it abstracting the busy work out of sight within the GranCloud objects.

I divided the synthesis model into three core components: a fully customizable grain SynthDef object acting as a grain synthesis template, a control data repository storing objects defining how grain parameter values change over time, and a scheduling engine that creates grain synthesis nodes on the server using parameter values retrieved from the repository. Figure 5 is an illustration of the basic GranCloud architectural design. It shows how the various components work together on the SuperCollider client application to produce granular synthesis on the server.

Following this architectural model, the grain SynthDef is defined and sent to the server application external to the cloud object. This allows it to be fully customized to the needs of the synthesis and desires of the composer. The control data repository is stored within the GranCloud object to improve code

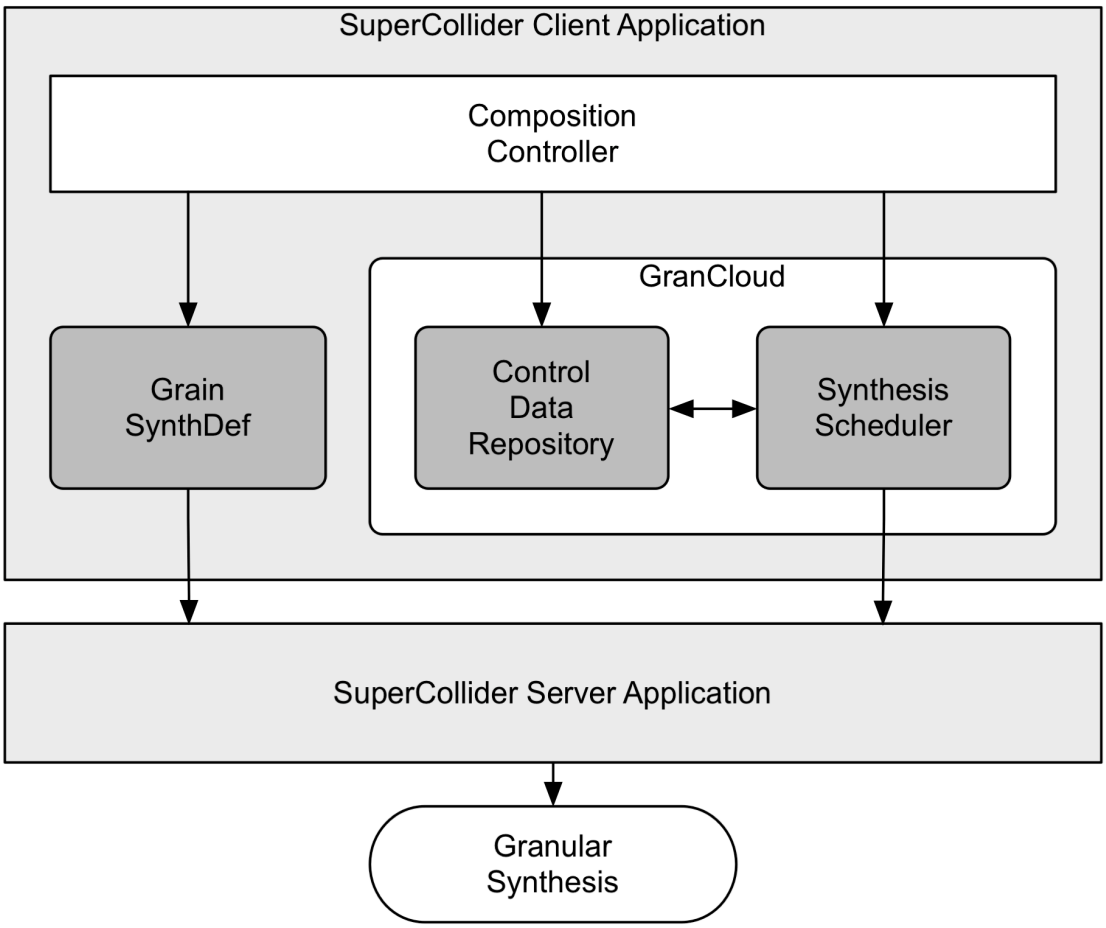


Fig. 5. GranCloud architectural design.

manageability and to provide easy access to the data by the scheduling engine. The parameter objects in the repository abstract the calculation details from the composition code. The synthesis scheduling engine is contained within the object methods of the cloud class improving the manageability of the controller code.

The Grain SynthDef

Each type of grain rendered on the server requires a SynthDef object acting as a template to define the synthesis and processing required for the generation of the grain. Designing SynthDefs is a common task performed by SuperCollider programmers. The SuperCollider client application provides a simple interface for creating SynthDefs and sending them to the server in preparation for audio generation. Once a grain SynthDef object has been defined in the client and sent to the server, the server is prepared to generate grains of that type on demand.

When considering the design requirements of the project, I decided to let users design their own grain SynthDefs and send them to the server themselves. There were multiple reasons for this decision. First, allowing users to define their own SynthDefs gives them the flexibility to work with any granular synthesis structure they choose. They may name their own parameters and add as many parameters as the grain synthesis requires. They may use any sound source for the granulation, apply any envelope, and perform any type of signal processing on a grain-by-grain basis according to their own desires and specific needs.

Another important reason for leaving SynthDef management outside of the cloud objects is that sending a SynthDef to the server and compiling it into a graph is a processor intensive task. It is best handled before the composition actually begins, reserving precious CPU cycles during performance for audio

processing. Since users typically send a large group non-granular SynthDefs to the server before any sound generation is performed, it made sense to allow them to send the grain SynthDefs at the same time. If SynthDef management had been incorporated within the cloud objects, then users would be required to either pre-define all of their clouds at the very beginning of the controller script, or else they would have to allow grain SynthDefs to be compiled on the server asynchronously while composition audio is being processed. The first option would use up a great deal of extra memory, while the second could potentially cause glitches in audio production when the CPU us heavily burdened.

SynthDef graph functions define named parameters that allow numeric data to be passed into synthesis nodes when they are created. The parameter values can be used to control various characteristics of the grain, such as frequency, duration, amplitude, sound source, envelope shape, and other control variables needed for the implementation. Users may define as many parameters as they need. They may also choose their own names for them, as long as they use the same names when parameter objects set in the control data repository.

The only absolute requirement GranCloud asserts for grain SynthDef design is that grain synthesis nodes must be able to remove themselves from the server once audio processing for the grain is complete. While this may seem like a serious limitation, it was a design decision that improved the processing efficiency of the application considerably. Enforcement of the requirement allows the synthesis scheduler to focus exclusively on event generation, saving

considerable computation, scheduling, and messaging that would be required to perform explicit node destruction as well. Since several methods for synthesis node self-destruction are built into many commonly used unit generators, I did not consider the requirement an unreasonable assertion.

An example of controller code defining a simple grain SynthDef object is displayed in figure 6. This example was used in *Creo* to define a simple grain based on an enveloped sine wave.

```
SynthDef(\sine_grain, { arg out=0, dur=0.1, amp=0.2,
        freq=440,envAttack=0.5, pan=0;
    var env, audio;
    env = EnvGen.ar(
        Env([0,1,0], [envAttack, 1-envAttack], \sine),
        1, amp, 0, dur, doneAction: 2
    );
    audio = SinOsc.ar(freq, 0, amp) * env;
    audio = Pan2.ar(audio, pan);
    Out.ar(out, audio);
});
```

Fig. 6. Code example showing the creation of a SynthDef named *sine_grain.*

The example code defines a SynthDef named *sine_grain* that uses six parameters to control various characteristics of the grain. Default values are provided for the parameters in case they are not specified in the cloud object controlling the grains. A quasi-Gaussian envelope object with a variable peak location is converted into a signal using an EnvGen unit generator. The EnvGen doneAction parameter is set to 2, instructing it to free the synthesis node from the

server once the end of the envelope has been reached. A sine oscillator is used as a sound source and the envelope applied to the sine wave. The resulting sound is processed using a panning unit generator and the resulting audio signal written to adjacent audio busses indicated by the value of the *out* parameter.

Three plots of grain audio signals generated by the *sine_grain* example are shown in figure 7. The differing shapes were created using the same SynthDef by changing the values of the *freq* and *envAttack* parameters.

freq = 440
envAttack = 0.5

freq = 330
envAttack = 0.1

freq = 880
envAttack = 0.9

Fig. 7. Grain waveforms generated using the sine_grain SynthDef.

The Control Data Repository

The control data repository is a central storage area within GranCloud objects where all of the control data necessary to generate a granular cloud is kept. The repository is not a single entity, but rather a set of instance variables within cloud objects that store data objects representing specific types of control

data. The cloud objects provide a set of convenience methods to assist in the management of data within the repository.

The repository contains variables to store three basic types of information: configuration variables identifying the server, grain SynthDef name, cloud playback duration, and a possible delay to playback initialization; routing variables controlling node order-of-execution and audio signal bussing; and an array of grain parameter objects defining how grain parameter values change over time.

Table 1 contains a list of names of the GranCloud instance variables used to store repository information. The list also indicates the type of data object stored in each variable and a brief description.

Table 1. GranCloud Instance Variables Used for the Control Data Repository

Instance Variable	Object Type	Description
server	Server	Identifies location of SuperCollider Server to use.
def	Symbol or String	Stores the grain SynthDef name or a function or stream that generates SynthDef names for individual grains.
duration	Number or inf	Indicates a period of time in seconds defining how long playback will render before automatically stopping. A value of inf (infinity) will cause audio generation to continue indefinitely until manually stopped.

Table 1. (continued)

Instance Variable	Object Type	Description
delay	Number	Indicates a period in seconds that sound generation will delay after the play method has been initiated. This is useful when clouds that start at different times are synchronized in groups.
dur	Integer	An integer indicating the audio bus to which grain audio signals will be routed.
out	Integer	An integer indicating the starting audio bus channel to which the audio signal will be written.
out2	Integer	An integer indicating a second audio bus to which grain audio signals will be routed. If null, no secondary copy will be routed.
target	Node	Indicates a target synthesis node for playback to which the addAction refers for controlling order-of-execution.
addAction	Symbol	Standard SuperCollider addAction indicating where the grain nodes should be placed in order of execution relative to the target node.
stopAction	Function	A function to execute once playback has completely stopped.
params	Array	An array of parameter objects defining grain parameter values and how they change over time.
paramNames	Array	An array of parameter names matching index numbers of parameter objects in the params variable.

The most significant repository variable for the grain synthesis is the *params* variable. The *params* variable stores an array of GranCloud parameter objects, each representing one of the grain parameters. Each object contains

control data defining the values of a parameter change over the life of the cloud. The parameter names corresponding to each object are stored in a parallel array in the *paramNames* instance variable.

I created several different types of parameter objects in order to represent many different methods of defining and generating the values. Some parameter objects use algorithms to convert control data to discrete values. Others map parameter values to different types of internal or external control sources.

All parameter objects implement a common interface to request discrete values at any given point in time over the existence of the cloud. Each object is a subclass of the SuperCollider Stream class, which serves as the base model for the interface. Stream objects represent a continuous sequence of values that may be retrieved one at a time by calling the *next* method multiple times on the object. Each time the *next* method is called, the next value in the sequence is returned.

Since many parameter objects are time dependent, the *next* method is passed a *time* parameter to retrieve values specific to that time within the existence of the cloud. The *time* value used represents the time that has elapsed since the start of the cloud measured in seconds. The *next* method may also be passed a parameter named *grainArgs* that contains a Dictionary of all parameter values for a grain that have already been retrieved for the current grain. This allows parameter calculations to be based on the values of other grain parameters if desired.

Some parameters do not rely on discrete calculated values. Instead, they are mapped directly to specific internal or external control sources. Parameter objects representing these will return a null value for the *next* method, but implement a *map* method that returns the control bus map message needed to connect the parameter input to the appropriate bus.

Some parameter objects define a graph function for a synthesis node. These nodes generate control signals that are mapped to the parameter input represented by the object. Objects of this sort implement a *controlMsg* method in addition to the *map* message. The *controlMsg* method returns a synthesis node creation message to send to the server with the first grain of the cloud.

Table 2 contains a list of the different parameter object classes currently provided with GranCloud. This list does not contain all of the ways that control data may be represented and generated for granular synthesis, but contains the most commonly used methods. Since parameter objects store control data internally and use a common interface for the retrieval of that data, the synthesis scheduler does not need to know anything about the object types stored in the repository. It simply calls the *controlMsg, next,* and *map* methods on each object to retrieve the appropriate values and messages for the next grain. This design also allows users to custom parameter object classes as long as they implement the same methods for data retrieval. A more detailed discussion of how each parameter object works has been included within the *Parameter Object Classes* section later in this document.

Table 2. Parameter Object Classes Included with GranCloud

Parameter Class	Control Data	Description
GCValue	A single data object representing discrete values.	Discrete values are retrieved from a single data object. The object may be a static number, a collection of numbers, a function that returns numbers, a stream that returns numbers, or an Env object representing values that change over time.
GCMinMax	Two data objects representing the min/max bounds of a distribution, and a function to generate values within the distribution.	Discrete values are generated by a distribution function (random or not) within a range defined by data objects representing a minimum and maximum boundary. The same types of data objects possible for GCValue objects may be used to define the minimum and maximum.
GCCenterDev	Two objects representing a center value +/- a deviation from the center, and a function generating the deviation.	Generates discrete values by adding a calculated deviation amount (random or not) to a center value. The same types of data objects possible for GCValue objects may be used to define the center and deviation.
GCControlMap	A graph function defining a control synthesis node.	Maps a parameter input directly to the output of an internally defined control synthesis node.
GCTrajectoryMap	A Trajectory object.	Maps x, y, and z input parameters to the outputs of a Trajectory synthesis node.
GCBusMap	A control bus object.	Maps a parameter input directly to a control bus.
GCMidiMap	A MIDI channel and event type	Maps a parameter value to a MIDI channel and event type.

The parameter objects are contained within a standard SuperCollider Array object in the *params* instance variable. Array objects store an ordered collection of objects that are accessible by index numbers. Using an Array object allows any number of parameter objects to be stored. It allows parameter values to be calculated in a specific order, which is important if some parameter values are used in the calculation of others. Array objects are also easy and efficient for iteration purposes, when a common process needs to be performed on each object.

Array objects are not, however, the easiest object to manage within code since programmers must keep in mind what objects are associated with each index. Since parameter objects are named, using a Dictionary[14] object would make more sense for accessibility. Then users could access specific parameter objects by name instead of index. This would eliminate the need for the *paramNames* array, since the dictionary would already have the names; however, another method of specifying processing order would be required. In addition iteration through a Dictionary object is less efficient than Array iteration. Since every grain in a granular cloud requires iteration through the parameter objects, and since several hundred grains may be generated each second, I decided to use the more efficient Array object and provide some Dictionary-like convenience methods to simplify accessibility.

[14] Dictionary objects store an unordered collection of objects that can be reference by name rather than index.

The *at* method is provided in cloud classes as a convenience method for accessing parameter objects by name. The method may receive a numeric or parameter name as a parameter. If the name is passed, the method converts the name to index internally to retrieve the associated parameter object. If an index is passed, it retrieves the object from the array directly.

The *put* method is a mutator method that allows parameter objects to be added to or replaced in the *params* array by name. When called, the method is passed a parameter name and object. If the method finds a parameter object with the same name, it is replaced in the same position of the array. If a matching parameter object does not already exist, the parameter object and name are added to the end of the *params* and *paramNames* arrays respectively.

The *at* and *put* methods also enable users to utilize built-in syntax shortcuts in the SuperCollider client language that allow values within a collection to be accessed or set using a square bracket notation. Using this syntax, users may access parameter objects by placing the name in square brackets immediately following the cloud object variable name (e.g. cloud[\rate]).

It is often very convenient to define grain parameters values using a single static SuperCollider object. Examples include static numbers, functions generating number, envelope objects, or stream objects. Objects of these types may be wrapped within GCValue objects in order to be treated as a parameter object. Since this is such a common method of specifying data, the *put* method will detect the object type and wrap raw data objects in a GCValue object

automatically. This shortcut saves composers an extra step when defining grain parameters represented by single data objects. Figure 8 shows code examples of the *at* and *put* methods used with and without syntax shortcuts.

```
// The cloud variable contains a cloud object.
cloud = GranCloudSimple.new();

// The put method called using full syntax and
// shortcut. Both lines are equivalent.
cloud.put(\rate, GCMinMax(0.005, 0.015));
cloud[\rate] = GCMinMax(0.005, 0.015);

// The put method with GCValue vs. raw data object.
// Both lines are equivalent.
// 440 is converted to GCValue(440) in the second line
cloud[\freq] = GCValue(440);
cloud[\freq] = 440;

// The at method called by name and index with and
// without syntax shortcut. All three lines are
// equivalent if rate is the first param object.
rate = cloud.at(\rate);
rate = cloud[\rate];
rate = cloud.at(0);
rate = cloud[0]
```

Fig. 8. Parameter object access and mutation using the *at* and *put* methods.

The Synthesis Scheduling Engine

The GranCloud synthesis scheduling engine is contained within the *prepare*, *play,* and *stop* methods of the cloud classes. It is responsible for preparing, instantiating, and controlling all of the synthesis required to generate the granular cloud. It retrieves discrete parameter values and server messages

for each grain from the control data repository and generates all of the OSC messaging needed to create and control the grain nodes on the server.

The *prepare* method must be called prior to sound generation to prepare the server for the synthesis. The method first validates required bussing and routing parameters and applies common defaults to any unset values. It then creates a group node on the server that will contain all of the grain synthesis nodes for the cloud. The group node is placed in a position on the node tree defined by the *target* and *addAction* variables of control data repository to ensure proper order-of-execution of the grain nodes. Finally, the method executes the *prepare* method of each grain parameter object. Many of the parameter objects do not require preparation and contain empty *prepare* methods that do nothing. Other parameter objects use the *prepare* method to send custom SynthDefs to the server for synthesis nodes that generate control signals. Those control signals are later mapped directly to the grain parameter inputs represented by the parameter object.

If GCControlMap or GCTrajectoryMap parameter objects are used in a cloud, the *play* method should not be called immediately following the *prepare* method. These objects send control SynthDefs to the server. Compiling a SynthDef to a graph is an asynchronous process. If the graphs have not been compiled before the first grain nodes execute, the corresponding synthesis control node cannot be created and grain parameters will be mapped to empty busses. Allowing one second between execution of the *prepare* and *play*

methods is usually a sufficient delay to allow the graphs to compile. As a failsafe, the *play* method will execute the *prepare* method automatically if it detects it has not been called. This allows users to omit explicitly calling the *prepare* method if they do not use GCControlMap or GCTrajectoryMap parameter objects.

The actual grain scheduling logic is contained within the *play* method. A flowchart showing the basic logic of the scheduler is displayed in figure 9. The *play* method may be passed a single parameter indicating an initial start time. The start time parameter allows users to start the cloud playing in the middle of the cloud with respect to time. It defaults to zero and defines the number of seconds within normal playback that the cloud should begin execution.

The *play* method first validates parameters in the repository. Suitable defaults are set for many of the routing and configuration variables if explicit values are not provided.

The project design purposefully left SynthDef design and parameter names fully customizable; however, the scheduling engine must know the grain rate and duration parameters in order to initiate grains and remove synthesis control nodes at the proper times. The scheduler requires a parameter object named *rate* defining the rate at which grains are initiated. The value for the *rate* parameter specifically defines the time in seconds between the start of each subsequent grain. The grain duration must be specified in a parameter named *dur.* This allows the scheduler to track when the last grain has completed execution so it can know when to release control synthesis nodes.

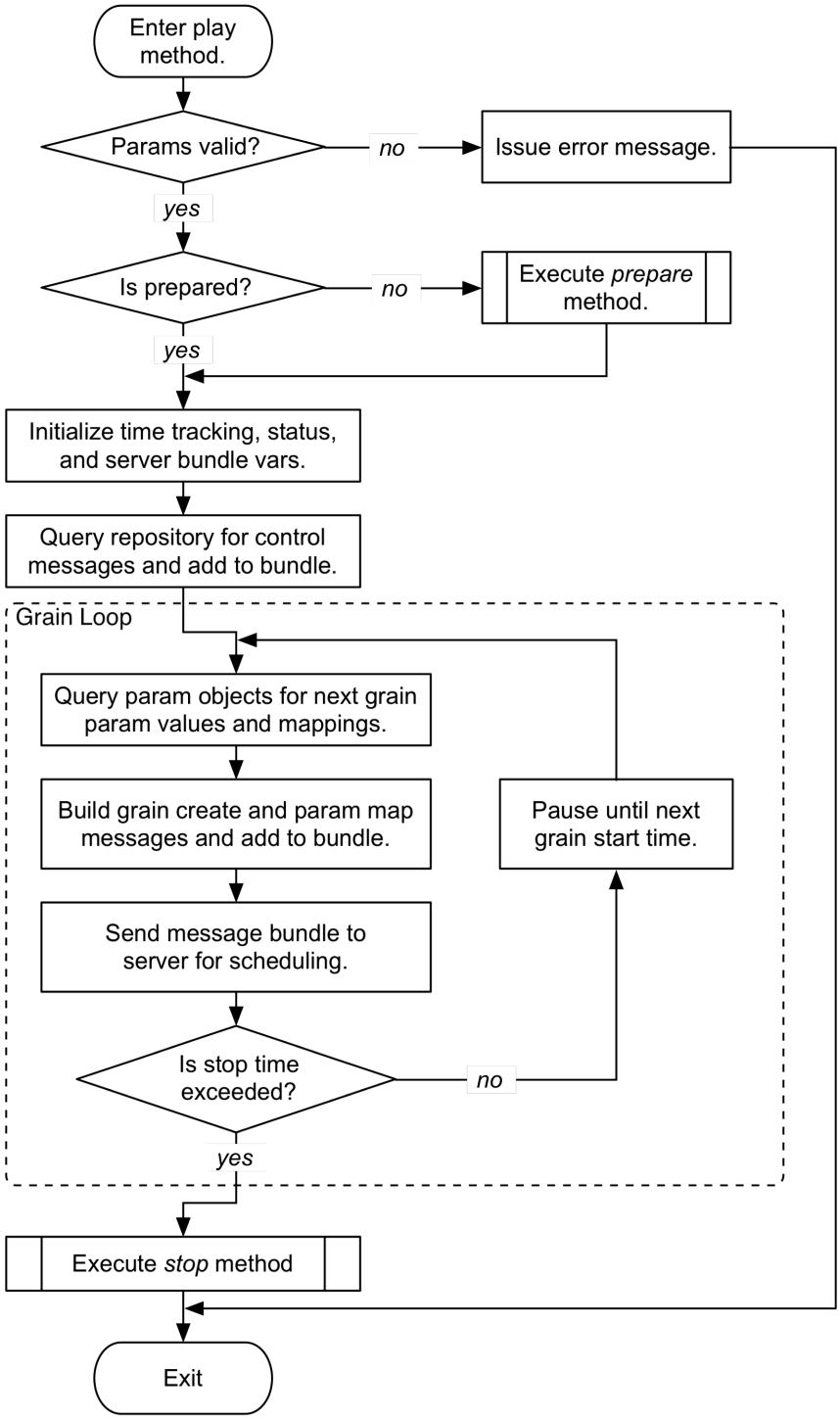


Fig. 9. GranCloud synthesis scheduling engine flowchart.

Once parameters have been validated, the *play* method initializes a set if internal variables used to track the timing and playback status of the cloud while it is being rendered. The scheduler uses an internal system clock as a timing reference and four time-related instance variables to track time for scheduling. The clock has a method named *seconds* that can be used to obtain the application time—a number indicating the time in seconds since the application was booted.

When the play method is called, a snapshot of the application time is taken and stored in the *startTime* instance variable. All subsequent timing is then calculated relative to the *startTime.* A *playTime* variable represents how long the cloud has been playing. During playback this variable is updated each time it is needed by subtracting the current application time from the *startTime.* A third time parameter, *endTime,* represents the time after which the cloud should stop playing. This value is calculated by adding the cloud duration to the *startTime*. It is calculated and stored ahead of time to minimize redundant processing within the grain loop. A fourth time parameter, *nextGrainTime,* represents the *playTime* at which the next grain node should begin execution on the server. It is derived from the grain *rate* parameter value for each grain and adjusted slightly to correct a time drift created by the CPU processing required to make grain parameter calculations and traverse the grain loop.

After the timing variables are initialized for the first time, the scheduler sets the playback status variables. These variables are named *isPlaying, isPaused,*

and *isStopped.* They contain Boolean values indicating whether the cloud is playing, paused, and/or stopped respectively. One might question why both an *isPlaying* and *isStopped* Boolean is required when they seem to be directly exclusive of each other. Both are necessary because a short time may exist when the rendering engine is no longer set to be playing, but synthesis has not completely stopped due to previously scheduled grains that are completing execution on the server. When the scheduled grains have completed, the *isStopped* variable is set to *true.*

Once the timing and status variables are set, the scheduler calls an internal method named *prepareGrain* to build a bundle of server messages required to generate the first grain. This method handles all interaction with the parameter objects necessary to retrieve control node creation messages, explicit grain parameter values, and bus mappings for each grain. Each time the method is called, it is passed the *nextGrainTime* value and a Boolean flag indicating whether this is the first grain or not. If it is the first grain, the message bundle returned will include any control synthesis node creation messages needed by the cloud. The *nextGrainTime* parameter is passed on to the parameter objects when methods are called to retrieve server messages and parameter values for that particular point in time. The flowchart in figure 10 shows the core logic used within the *prepareGrain* method.

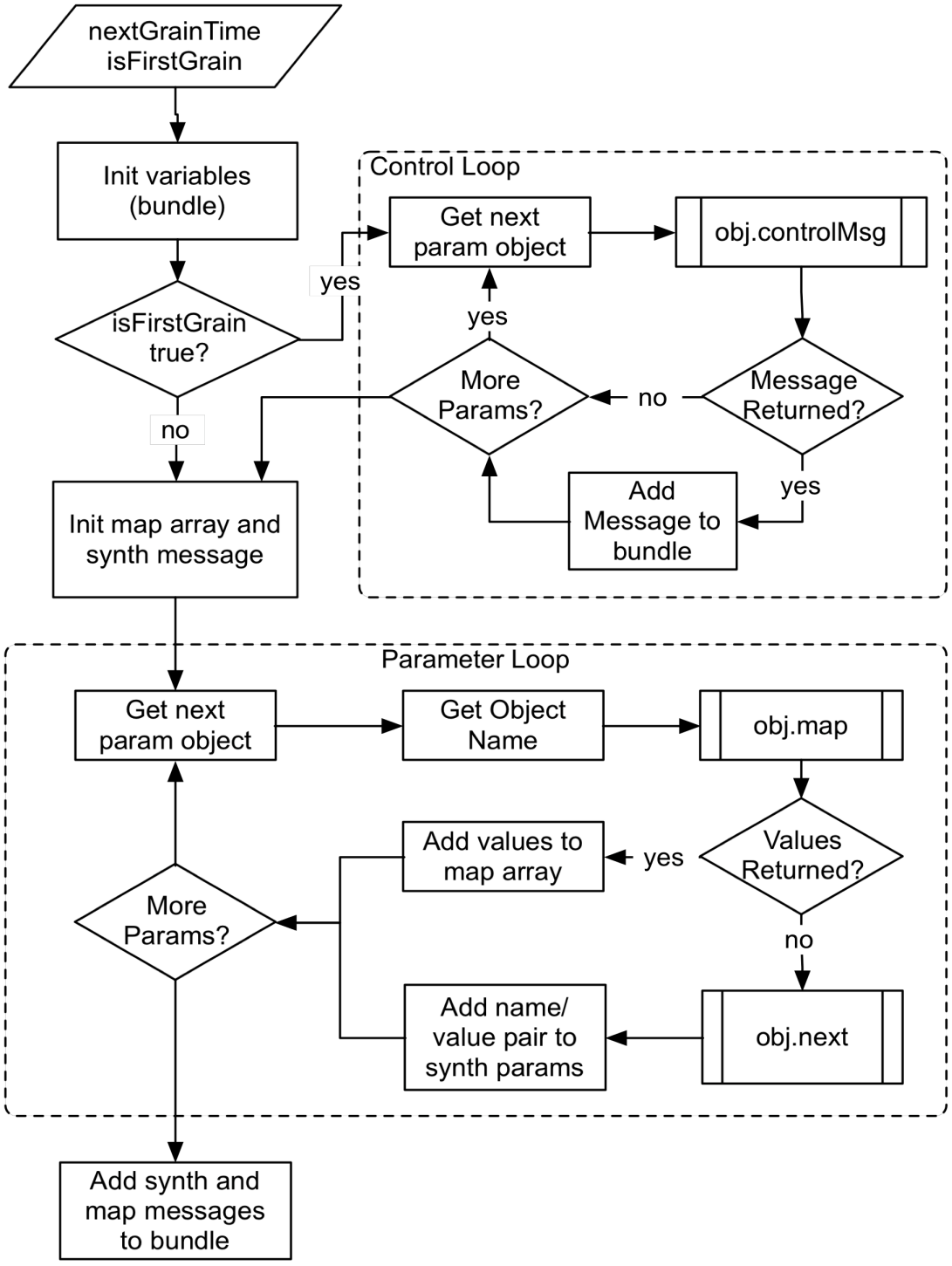


Fig. 10. GranCloud *prepareGrain* method flowchart.

When the *prepareGrain* method is called, if the *isFirstGrain* Boolean is true, then the method calls the *controMsg* method on each grain parameter to gather any server messages required by parameter objects for control synthesis nodes that will be mapped to parameter inputs. The control messages are grouped together in a bundle that will be scheduled to execute synchronously with the first grain node.

Next, the method prepares an Array object to contain the values required to build the grain node creation message. The array is initialized with the SynthDef name to use for the grain and various routing parameters controlling the order of execution. The SynthDef name is retrieved from the *def* variable of the control data repository. This variable may contain either a single SynthDef name, or an object that generates SynthDef names when the *next* or *value* method is called on the object.

The *prepareGrain* method then adds name/value pairs to the message array for grain parameters. First, a parameter named *out* is added with the value stored in the *out* instance variable of the cloud object. If the *out2* instance variable has been populated, it is added as an *out2* parameter. These parameters specify the audio bus numbers to which the grain audio will be routed.

Next, the method iterates through the parameter objects stored in the *params* instance variable to calculate grain parameter values and bus mappings. For each parameter object, the *map* method is first called to see if any bus

mapping messages are needed for the parameter. If bus-mapping messages are returned, they are collected in an array to be added to the message bundle. If no messages are returned, then the *next* method is called on the parameter object to calculate and return an explicit value to use for the grain parameter. These values are collected and added to the grain node creation message as a name/value pair. The *next* method receives the *nextGrainTime* and a Dictionary object containing the values of all previously calculated parameters in case those values are needed for the algorithm calculating the value.

Once the parameter iteration is complete, the synthesis node creation message and collected bus mapping messages are added to the server message bundle. The *prepareGrain* method is finished and execution returns to the *play* method.

Another feature of the *prepareGrain* method is its ability to detect if a parameter object returns an explicit null value. Stream objects return an explicit null value when the end of the stream of values has been reached. Users who program with the Pattern and Stream event interface in SuperCollider are used a null value stopping execution of events. To remain consistent with this behavior, the *prepareGrain* method executes the cloud *stop* method if any parameter object returns an explicit null value.

Once the first grain has been prepared, the *play* method enters the synthesis scheduling loop. The code within the scheduling loop first checks if the cloud has been stopped or paused by checking the value of the *isPlaying*

variable. If the Boolean is false, execution exists the scheduling loop. If it is true, execution sends the prepared message bundle generated by the *prepareGrain* method to the server.

The scheduler then calculates the *nextGrainTime,* stores the completion time of the last sounding grain that has been scheduled in a temporary variable, and calls the *prepareGrain* method with a false *isFirstGrain* parameter to generate the message bundle for the next grain. At the end of the loop, the *play* method checks the *playTime* to ensure it is not past the *endTime* of the cloud. If it is past the *endTime,* the *stop* method is called. If the *endTime* is not past, the grain loop pauses execution until the scheduled time for the next grain execution and repeats the scheduling loop.

The *stop* method may be called explicitly on the cloud object or internally by the synthesis engine. The *stop* method sets the *isPlaying* method to false so the scheduler will exit the scheduling loop on the next pass. Then, it waits until the last sounding grain that has been scheduled finishes execution and frees the group node and control synthesis nodes from the server. Once the nodes have been removed, the *isStopped* Boolean is set to *true* and an optional *stopAction* function executed if one has been defined for the cloud. The *stopAction* function can be used to trigger external tasks in a composition that are dependent on a cloud finishing execution.

All of this scheduling logic is abstracted within the cloud objects, so none of it needs to be included within composition controller scripts using them. The

controllers simply instantiate the cloud objects, populate them with control data, and call the playback methods at the appropriate times to execute the synthesis. This abstraction removed a tremendous amount of redundant code from the controller script used by *Creo* making the code much cleaner and easier to manage.

CHAPTER 3

GranCloud: Class Structure

The classes in the GranCloud project are structured in an object-oriented hierarchy designed to minimize the need for redundant code to perform tasks common to multiple objects. Some classes are abstract parent classes that are not meant to be instantiated directly. They provide singular locations to define code common to multiple subclasses of the parent to avoid the need for code duplication in each child class. Other classes are non-abstract classes that are intended to be used to instantiate actual objects. In some cases, classes subclass non-abstract classes to extend the functionality of the parent.

Cloud Object Classes

The cloud classes are the main controller classes of the GranCloud project. Each object instance of a cloud class represents a specific granular cloud. Cloud objects contain all the control data necessary to define the characteristics of the clouds they represent as well as all of the methods needed to schedule the synthesis on the server.

Class Hierarchy

The GranCloud project includes two cloud classes: GranCloudSimple, and GranCloud2. The GranCloudSimple class is a cloud class that contains all the instance variables and methods needed to generate the synthesis. It is a simple class that focuses exclusively on the requirements of sound generation. It may

be instantiated and used directly if the extended functionality of the GranCloud2 object is not required. It contains the control data repository and synthesis scheduling engines, making it the central core of the entire project.

The GranCloud2 class is a subclass of the GranCloudSimple class. As a subclass, it contains all of the features of the parent class and adds additional functionality. The GranCloud2 class adds support to allow clouds to be executed together in synchronous groups. It also adds support for various features used in graphical interfaces that may be used to visualize and manipulate cloud parameter data. In addition, it allows grain SynthDefs to be defined within the object either as a GrainBuilder object or a unit generator graph function. In either case the GranCloud2 object will automatically build the SynthDef object from the stored objects and send it to the server when the *prepare* method is called. Table 3 contains a summary of the class inheritance hierarchy of the cloud objects.

Table 3. Cloud Class Inheritance and Descriptions

Class Name	Parent Object	Description
GranCloudSimple	Object	A simple cloud class encapsulating only the basic functionality needed for the generation of granular synthesis.
GranCloud2	GranCloudSimple	A cloud class with added functionality used by graphical interfaces and external control classes.

GranCloudSimple

The GranCloudSimple object contains the control data repository and synthesis scheduling engine. It provides all the core functionality needed for the sound generation. Table 4 lists the instance variables contained within the object and summarizes the purpose of each. Table 5 shows descriptions of the methods that may be called on the object.

Table 4. Instance Variables of the GranCloudSimple Class

Instance Variable	Data Type	Description
<>server	Server	An SC Server object identifying the server to use for synthesis. Defaults to the default server set for the client application if not set.
<>def	String, Symbol, or Function	The grain SynthDef name as a String or Symbol, or a function that returns grain SynthDef names when evaluated for each grain.
<>duration	Number	The duration of the cloud in seconds. May use inf for infinite duration. Defaults to inf.
<>delay	Number	Delays the start of playback by this number of seconds after the play method is executed. Defaults to 0.
<>out	Integer	The primary output audio bus index sent to grains in the out parameter. Defaults to 0.
<>out2	Integer	A secondary output audio bus index sent to grains in the out2 parameter if a value is set.

Table 4. (continued)

Instance Variable	Data Type	Description
<>target	Node	A target Group, Synth, or Server object to reference for setting up the order-of-execution for the grain nodes.
<>addAction	Symbol	Specifies where grain nodes should be placed in order-of-execution on the node tree relative to the target node. Possible values are: \addToHead, \addToTail, \addBefore, \addAfter, or \addReplace.
<>stopAction	Function	A function that will be executed when the cloud has stopped playing either due to the stop method being called or exceeding the duration of the cloud.
<params	Array	An array of parameter objects defining grain parameter control data. All grain parameters not using a default value should have a parameter object in this array representing how parameter values should change over time. Parameter values are calculated in the order they appear in this array.
<paramNames	Array	A parallel array to the params array containing the parameter names that parameter objects represent.
<>timeStretch	Integer	A time multiplier that can be used to stretch or compress the passage of time relative to the parameter control data. For example, setting to 2 will move through the data twice as slow. Setting to 0.5 will move through the data twice as fast.

Table 4. (continued)

Instance Variable	Data Type	Description
<>scrub	Boolean	A Boolean that turns on scrub functionality allowing time within the cloud to be controlled by an external source, rather than by the internal clock. This allows an external source to “scrub” through the playback of the cloud by controlling the playTime variable directly.
<isPlaying	Boolean	Indicates if the cloud is actively playing or not.
<isPaused	Boolean	Indicates if the cloud is currently paused or not.
<isStopped	Boolean	Indicates if the cloud is currently stopped (not playing, and no grains from previous playback are still sounding).
<isPrepared	Boolean	Indicates if the prepare method has been called or not.
<>playTime	Number	A number indicating how many seconds into playback time has progressed. Under normal playback this will indicate how long the cloud has been playing, but variables that adjust time (timeStretch and scrub) will change that.
<>startTime	Number	Stores the System time (SystemClock.seconds) when playback started.
endTime	Number	Indicates the System time at which the last sounding grain currently scheduled will complete execution.
nextGrainTime	Number	Indicates the start time for the next grain to be scheduled. Should match the sum of each previously scheduled grain’s rate parameter values.

Table 4. (continued)

Instance Variable	Data Type	Description
<grainArgs	IdentityDictionary	A Dictionary object containing the most recently calculated grain parameter values keyed by parameter name.
<mapMsg	Array	A private variable used by the scheduler for collecting bus mappings for grains.
<group	Group	A private variable storing a Group object representing the group node on the server where all grains are placed on the node tree.
<bundle	Array	A private variable storing an array for collecting groups of server messages that should be executed simultaneously as a bundle for grain execution.

Note: In all instance variable tables, a less-than sign (<) before the instance variable name indicates the variable has a getter method defined that allows users to access the variable from outside of the object. A greater-than sign (>) before the instance variable name indicates the variable has a setter method that allows users to change the value of the variable. Both signs may be present to indicate the object supports both a getter and setter for the variable.

Table 5. Methods and Parameter Descriptions of the GranCloudSimple Class

Method Name	Method Description	
	Param. Name	Parameter Description
*new	Instantiates a new GranCloudSimple object.	
	server def duration delay out out2 target addAction stopAction params timeStretch scrub	These parameters may be used to initialize instance variables of the same name. See Table 4 for a description of each.
init	Private method called by *new method to initialize variables and defaults.	
at	Gets a parameter by index or name.	
	index	Either a numeric index into the params array or the name of a parameter object to return.
put	Set a parameter object in the params array by name. If an object already exists by that name, the object will be replaced. Otherwise the object will be added to the end of the array.	
	name	The name of the parameter object to place.
	value	A parameter object, or a single raw data object (number, envelope, stream, etc…) to be wrapped in a GCValue object and placed in the params array.

Table 5. (continued)

Method Name	Method Description	
	Param. Name	Parameter Description
add	Adds a parameter object to the end of the params array.	
	name	The name of the parameter object to add.
	value	The parameter object to add.
removeAt	Removes an object from the params array by name or index.	
	index	A numeric index or name of a parameter object to remove from the cloud.
reset	Sets the playTime of the cloud to the specified time and resets all parameter objects to be able to be rendered again.	
	time	The new time to which the playTime variable will be set, specified in seconds. Defaults to 0.
prepare	Prepares a cloud for playback, setting defaults for any unset configuration variables and creating the grain group node on the server.	
play	Starts playback of the scheduling engine at the specified time in the cloud.	
	time	Specifies the playTime within the cloud control data to begin playback. Defaults to 0.
pause	Used to pause a playing cloud or restart playback of a paused cloud.	
stop	Stops playback of the scheduling engine.	
prepareGrain	Private method for calculating the parameters for the next grain at the specified time.	
	time	The playTime within the cloud that should be used for the parameter calculations.
	firstGrain	A Boolean indicating whether this is the first grain of the cloud or not

Table 5. (continued)

Method Name	Method Description	
	Param. Name	Parameter Description
free	A synonym for the stop method.	
writeArchive	Writes the cloud and all control data to a text-file archive that can be read back into memory at a later time using the *readArchive method.	
	path	A file path specifying the location and filename to use to store the archive.
*readArchive	Reads an archive from a text-file and instantiates the cloud object based on the archive.	
	path	Specifies the location and filename of the archive.
	server	Since Server objects cannot be archived, this is used to specify Server to use in the cloud.
	target	Since target node object cannot be archived, this is used to reset the target of the archived cloud.

Note: In all Method Description tables, an asterisk in front of the method name indicates it is a class method called on the class, rather than an instance method called on an object instance of the class.

GranCloud2

GranCloud2 is a cloud class that is a subclass of the GranCloudSimple class. GranCloud2 objects include all of the instance variables and methods of the parent class along with extended the functionality as discussed above. Table 6 lists the additional instance variables used by the GranCloud2 class that are

not a part of the GranCloudSimple class. Table 7 summarizes the additional methods available that extend the functionality of the GranCloudSimple class.

Table 6. Instance Variables of the GranCloud2 Class Not Inherited from the GranCloudSimple Class

Instance Variable	Data Type	Description
<>name	String or Symbol	Stores a name for the cloud for reference in groups or in graphical interfaces.
<>playEnabled	Boolean	Indicates whether playback is specifically enabled for this cloud. This is used within GranCloudGroup objects to enable or disable specific clouds in the group.
<>playDisabled	Boolean	Indicates whether playback is specifically disabled for this cloud. Also used within GranCloudGroup objects
<>solo	Boolean	Indicates whether to cloud is set to be played exclusive of other clouds in a group.
<>level	Number	Indicates a dynamic level adjustment parameter included in grain creation messages to scale the overall audio output. This is used to mix clouds within groups without needing to change control data in the clouds.
<>grainBuilder	GrainBuilder	Allows a GrainBuilder object to be associated with the cloud. The GrainBuilder represents the grain SynthDef, providing support for several features allowing grain SynthDefs to be built using a graphical interface. If included the prepare method will automatically build and send the SynthDef represented by the GrainBuilder settings to the Server.

Table 6. (continued)

Instance Variable	Data Type	Description
<ugenGraphFunc	Ugen Graph Function	Stores a graph function defining the grain SynthDef. If included, the prepare method will compile it into a SynthDef and send it to the server automatically.

Table 7. Methods and Parameter Descriptions of the GranCloud2 Class Not Inherited from the GranCloudSimple Class

Method Name	Method Description	
	Param. Name	Parameter Description
prepare	The GranCloudSimple prepare method is expanded to send the grain SynthDef if the grainBuilder or ugenGraphFunc instance variable is set.	
play	The play method is expanded to check the playEnabled, playDisabled, and solo flags to determine whether to actually generate the cloud or not.	
	time	Specifies the playTime within the cloud control data to begin playback. Defaults to 0.
pause	The pause method is expanded to check the playEnabled, playDisabled, and solo flags to determine whether to actually respond to the call or not.	
stop	The stop method is expanded to check the playEnabled, playDisabled, and solo flags to determine whether it needs to stop the cloud or not.	
default	Sets enough default data within the cloud object to be able to actually render simple synthesis.	
defaultParams	Used to create default parameter objects allowing simple synthesis for a new cloud to be rendered.	

Table 7. (continued)

Method Name	Method Description	
	Param. Name	Parameter Description
defaultSpecs	Builds default ControlSpec objects for parameter objects that do not have them set. This allows clouds that are not fully filled with control data to be opened within a graphical interface.	

Parameter Object Classes

Parameter objects contain control data that define grain parameter values and how they change over time. The parameter objects are instantiated and stored in the *params* array of the cloud classes. Several types of parameter objects exist in order to allow composers to specify different calculation algorithms for grain parameters generation or to map parameters directly to external controls.

Parameter classes provided in the GranCloud project follow a naming convention in that they all begin with “GC.” This convention helps distinguish the parameter objects from other types of objects in the project.

Class Hierarchy

All Parameter objects inherit from the GCBase abstract parent class. The GCBase class contains logic that is common to all parameter classes and default declarations of methods that are required by the synthesis engine. Individual subclasses that require different logic than the default methods will override those methods in their own class definition. The GCBase class inherits from the

SuperCollider Stream class giving all parameter objects default Stream functionality.

The GCVaueBase class is an abstract parent for all parameter classes that return numeric parameter values for each grain. It provides instance variables and methods common to these classes. The GCMultiParam class is an abstract parent class for all GCValueBase subclasses that use more than one value object in their calculation algorithms.

The GCControlBase class is an abstract parent class for all parameter objects that map grain parameters to an external control source. It contains logic common to each class that maps parameter inputs to a bus instead of generating explicit values for each grain.

All other parameter objects are non-abstract classes that may be instantiated and used directly as parameter objects. Each non-abstract class has been discussed in its own subsection below. Table 8 shows the class hierarchy of the parameter objects and provides a basic description of each.

Table 8. Parameter Class Inheritance and Descriptions

Class Name	Parent Object	Description
GCBase	Stream	An abstract parent class containing default methods and code common to all parameter objects.

Table 8. (continued)

Class Name	Parent Object	Description
GCValueBase	GCBase	An abstract parent class containing code common to all parameter objects that return specific values for each grain.
GCValue	GCValueBase	A parameter class that generates values from a single raw data object (number, collection, function, envelope, or stream).
GCMultiParam	GCValueBase	An abstract parent class for objects calculating values from multiple raw data objects.
GCMinMax	GCMultiParam	A parameter object deriving values from a random distribution function bounded by two raw data objects representing the minimum and maximum value of the distribution.
GCCenterDev	GCMultiParam	A parameter object deriving random values calculated by adding or subtracting a random deviation from a center value. Raw data objects are stored to define the center and maximum deviation.
GCControlBase	GCBase	An abstract parameter class containing functionality common to parameter object that map parameter values to a control signal.
GCControlMap	GCControlBase	Maps a grain parameter to the output of a synthesis node generating a control signal. The SynthDef is built from a stored graph function and sent to the server automatically.

Table 8. (continued)

Class Name	Parent Object	Description
GCTrajectoryMap	GCControlBase	Maps a specific set of x, y, and z parameters to the output of a Trajectory synthesis node. Stores a Trajectory object that generates the control node that is mapped.
GCBusMap	GCControlBase	Maps the grain parameter to a specified control bus.
GCMidiMap	GCControlBase	Maps the grain to values coming from a specific MIDI source, channel, and event type.

Note: The indention of class names illustrates the hierarchical dependencies of the classes.

All non-abstract parameter classes implement a standard set of messages that the cloud object uses to retrieve data needed for grain parameters. Some additional common methods have been included to all to support key features needed for graphical interfaces. All parameter classes provide a *new* class method to instantiate an object and initialize the control data. Table 9 lists the methods common to all parameter objects and includes a brief description of each. Private methods that are never used directly by users are not included in this list. Users may define their own custom parameter object classes as long as they subclass the GCBase class and either implement these methods or inherit them from the parent.

Table 9. Methods Common to all Parameter Object Classes

Method Name	Method Description	
	Param. Name	Parameter Description
*new	A class method to instantiate a new parameter object. Parameters matching the instance variables of the method may be passed in to initialize those variables.	
init	A private method called by the *new method to initialize instance variables with defaults if values were not passed in to the new method.	
prepare	A method called by the scheduling engine when the cloud object is prepared to send SynthDefs defined by parameter objects to the Server.	
	Server	The Server object set in the cloud will be passed in when this method is called.
	Group	The object representing the grain Group node used for controlling order of execution is passed in when the method is called.
controlMsg	Called by the scheduling engine when the first grain parameters are retrieved. If any synthesis node initializations messages need to be executed with the first grain, they will be returned.	
next	Called by the scheduling engine for each grain to retrieve an explicit value for the parameter for that grain	
	time	Provides the playTime when the grain will be executed. This allows time-based parameter objects to return values change depending on the playTime.
	grainArgs	A Dictionary of parameter values that have already been calculated for this grain. This allows parameter values to be dependant on the values of other parameters.
map	Returns any bus mapping messages required to map the parameter input to a control bus.	

Table 9. (continued)

Method Name	Method Description	
	Param. Name	Parameter Description
	name	The name of the parameter since it will be needed to build the mapping message.
free	A method called by the scheduling engine when the cloud is stopped in order to free any control synthesis nodes that were initialized by grain parameter objects.	
reset	If the cloud reset method is called to reset the cloud back to its initial state, this method will be called on each parameter to reset stream objects to their original state.	
defaultSpecs	This method is called by the GranCloud2 defaultSpecs method in order to initialize default ControlSpec objects representing min/max boundaries for the range of the parameter values. ControlSpec objects are needed for graphical objects such as knobs and sliders need discrete min/max boundaries, whether the parameter object requires them or not.	
run	A method called by the pause and play methods of the cloud object in order to pause or resume control nodes. This is done by setting the run state of control nodes to true or false values.	
	bool	A Boolean value indicating the run state of control nodes. A true value starts the node running. A false value pauses the node.

GCValue

The GCValue class defines parameter objects that derive parameter values from a single raw data object. The *next* method is used to retrieve the next value from the data object. The *controlMsg* and *map* methods both return null values since no control nodes or bus mappings are required.

The data object used to represent values is stored in the *value* instance variable. The variable may contain an Env object, a Function object, or any type of SuperCollider object that returns a number when the *next* method is called. The most frequently used data objects include static numbers, Array or List objects containing numbers, a Function that returns numbers when evaluated, an Env (envelope) object, or a Stream returning discrete numbers when the *next* method is called. A second instance variable, named *spec,* allows users to assign a ContorolSpec object to the parameter. ControlSpec objects define minimum, maximum, and warp values for mapping parameter values to GUI objects such as sliders and knobs.

The GCValue object initializes a private instance variable named *valueMethod* that identifies the name of the method to use to request values from the data object. When a new GCValue object is instantiated, the *init* method detects the data object type and updates the *valueMethod* appropriately. If the type is a Function, the *value* method will be called on the data object. If it is an Env object, the *at* method will be used. The *next* method will be used for all other data types. In SuperCollider, Strings, Symbols, Numbers, and many other singular object types implement a *next* method that does nothing but return the object itself. This allows them to act as infinite streams that always return their own value.

When the *valueMethod* is executed on the data object, the *nextGrainTime* and *grainArgs* values are passed in as parameters, allowing the calculations to

be dependent on the grain *playTime* or on the values of previously calculated parameters. Figure 11 shows several examples of GCValue object instantiation using different data types.

```
// Instantiate using a static number.
// Returns an endless stream of 0.001.
obj = GCValue(0.001);

// Instantiate using a Stream Based on a Pattern.
// Returns an endless sequence 1, 2, 3, 1, 2, 3, etc…
obj = GCValue(Pseq([1, 2, 3], inf).asStream);

// Instantiate using an envelope object.
// Returns value of the Env at each grain start time.
obj = GCValue(Env([ 0, 1, 0.5, 0 ], [ 2, 3, 10 ]));

// Instantiate with a ControlSpec for GUI mapping.
obj = GCValue(10, ControlSpec(1, 20));
```

Fig. 11. Code examples of GCValue object instantiation using various data types.

GCMinMax

GCMinMax objects represent random distributions between a minimum and maximum value. Each time the *next* method is called to return a value, a distribution function is executed to retrieve a value between the minimum and maximum values. Users may define their own distribution function in a Function object that takes the minimum and maximum values as parameters, or they may specify the name a built-in SuperCollider binary function to use. Distributions may be random or not, depending on the function specified. If a distribution function is

not specified, it defaults to the built-in SuperCollider *rrand* function. The *rrand* function generates equally distributed random values between the minimum and maximum values.

The minimum, maximum, and distribution function values are stored in the *min, max,* and *dist* instance variables respectively. The *min* and *max* variables may contain the same data types that be used as the *value* in GCValue objects. This means they may be represented by static values or by values that change over time. The *dist* variable may contain the name of a built-in SuperCollider binary function, or it may contain a user-defined Function object.

The GCMinMax object also contains instance variables named *minSpec, maxSpec,* and *spec.* The *minSpec* and *maxSpec* variables store optional ControlSpec objects for mapping graphical objects to the *min* and *max* variables. The *spec* contains an optional ControlSpec object that can be used to constrain the output of the distribution calculation to the range of the ControlSpec.

When the *next* method is called on a GCMinMax object, discrete values are first retrieved from the *min* and *max* data objects in the same manner as the calculation of the *value* in the GCValue object. The distribution function is then evaluated passing in the discrete minimum and maximum values that were retrieved. Finally, if a *spec* object has been specified, the resulting value is constrained within the range of the *spec* object. Figure 12 shows several examples of the instantiation of GCMinMax objects.

```
// Instantiate object using a static min/max values.
// Returns random values between 10 and 20.
obj = GCMinMax(10, 20);

// Instantiate using a changing min and static max
// value. Returns random values between a min that
// changes from 1 to 10 over 30 seconds time, and a
// max value of 40.
obj = GCMinMax(Env([1, 10], [30]), 40);

// Use a built-in exponential distribution function.
obj = GCMinMax(10, 20, \exprand);

// Use a user-defined distribution function.
func = { arg min, max, time, params;
    [ min.rrand(max), min.rrand(max), min.rrand(max) ].sum / 3
};
obj = GCMinMax(1, 10, func);
```

Fig. 12. Code examples of the instantiation of GCMinMax using various objects.

GCCenterDev

GCCenterDev objects are very similar to GCMinMax objects. They both generate distributions; however, GCCenterDev objects define the distribution as a center value plus or minus a deviation from the center, rather than values within a specified minimum or maximum. The same types of data objects that GCValue objects accept may be used to define the center and maximum deviation values. A unary distribution function that takes the maximum deviation as a parameter value is used to generate actual deviation values that are applied to the center value to calculate the return value for the parameter. Like the GCMinMax object the distribution function may be a user-defined function or a

reference to a SuperCollider built-in function; however, this in GCCenterDev objects the function should be a unary function.

The *center, dev,* and *dist* instance variables store the center, maximum deviation, and distribution function values respectively. The object also contains instance variables named *centerSpec, devSpec,* and *spec* for containing optional ControlSpecs that may be mapped to graphic objects or constrain the resulting calculation in the same manner as the GCMinMax object.

When the *next* method is called on a GCCenterDev function, discrete center and maximum deviation values are first retrieved from the *center* and *dev* objects. Then the distribution function is executed using the maximum deviation value as a parameter to get the actual deviation for the grain. Similar to the GCMinMax object, user-defined functions are also passed the grain start time and a Dictionary of previously calculated parameters in case they need to be used in the calculation of the deviation. The actual calculated deviation is then added to the discrete center value and returned to the caller.

If a *spec* object has been specified, the return value will be constrained to the range of the ControlSpec. Figure 13 shows examples of the use of GCCenterDev objects.

```
// Instantiate using a center value that is scaled by
// a previously calculated \rate parameter with a
// static maximum dev value.
func = { arg dev, time, params; params[\rate] * 2 };
obj = GCCenterDev(func, 25);

// Instantiate using a built-in distribution function
obj = GCCenterDev(100, 30, \sum3rand);

// Instantiate using a user-defined function and
// ControlSpecs.
devSpec = ControlSpec(0, 100, \linear);
centerSpec = ControlSpec(20, 20000, \exp);
spec = ControlSpec(20, 20000, \exp);
func = { arg dev, time, params;  dev.rand2 * params[\rate] / time };
obj = GCCenterDev(
    Env([ 100, 3000 ], [ 30 ], \exponential),
    100,
    func,
    centerSpec,
    devSpec,
    spec
);
```

Fig. 13. Code examples of the instantiation GCCenterDev objects using various data types.

GCControlMap

A GCControlMap object patches the grain parameter input directly to the output of a control synthesis node via a control bus. This allows a grain parameter input to be controlled by a server generated signal such as a sine wave. The object contains either a unit generator graph function defining the control node or the name of an externally defined SynthDef to use as the control node. It also contains an optional array of argument name/value pairs to be included in the control node creation message.

If a unit generator graph function is specified, it is stored in the *ugenGraphFunc* instance variable. If that variable is not set, then it is expected that the *synthName* instance variable should contain the name of an external SynthDef that has already been sent to the server. The *args* instance variable contains the optional array of control node parameter values.

The GCControlMap object overrides the empty GCBase *prepare* method in order to build the *ugenGraphFunc* into a SynthDef, send it to the server, and allocate a control bus for the patch. The *prepare* method also builds the control node creation message that is returned when the *controlMsg* method is called by the scheduling engine.

The scheduling engine calls the *controlMsg* method and sends the node creation message to the server synchronously with the start of the first grain. The engine will then call the *map* method for each grain to retrieve the bus-mapping message needed to map the parameter input to the control signal. The *next* method will return a null value since the parameter is mapped to a bus instead of a discrete value.

The GCControlMap object also supports *run* and *free* methods to set the synthesis node play status in response to the *pause* and *play* methods and to free the node from the server once the cloud is stopped. Figure 14 shows how a GCControlMap object can be used to map a parameter input to a sine wave.

```
// Define a ugenGraphFunc and instantiate GCControlMap
// object.
graphFunc = {
    arg out=0, freq=20, phase=0, amp=1, add=0;
    Out.kr(out, SinOsc.kr(freq, phase, amp, add))
};
obj = GCControlMap(graphFunc, [\freq, 30, \amp, 60]);

// Same as above but using an external SynthDef.
SynthDef(\sine, {
    arg out=0, freq=20, phase=0, amp=1, add=0;
    Out.kr(out, SinOsc.kr(freq, phase, amp, add))
}).send(server);
obj = GCControlMap(nil, [\freq, 30, \amp, 60], \sine);
```

Fig. 14. Code example of a GCControlMap used to patch a parameter Input to a sine wave.

GCTrajectoryMap

Trajectory objects are a custom class of object I developed in order to define three-dimensional trajectories through space and time on an x, y, and z grid. Trajectory objects possess methods enabling them to function within a unit generator graph function to convert the defined trajectories into actual control signals for each axis.

The GCTrajectoryMap class allows composers to define a Trajectory object and map x, y, and z control signals from it to corresponding x, y, and z grain parameters. The GCTrajectoryMap is different than other parameter objects in that a single parameter object is used to supply values to three grain parameters. The object maps three related grain parameters to signals on a three-channel control bus. The grain parameter names used in the server

messaging will be derived from the parameter name concatenated with X, Y, or Z. For example, if the parameter is named *pan* in the control data repository, the GCTrajectoryMap will map the Trajectory signals from the control node to the *panX, panY, and panZ* grain parameter inputs.

The GCTrajectoryMap object stores the Trajectory object used to generate the signals in the *traj* instance variable. The object also contains instance variables named *specX, specY, and specZ* that may contain ControlSpec objects for use by graphical objects mapped to each axis.

The *prepare* method is important to the GCTrajectoryMap object in order to build the control node creation message returned by the *controlMsg* method. The *map* method returns the bus-mapping message that connects the grain input parameters to the trajectory signals outputs via a control bus. The *next, run,* and *free* methods are used in the same manner as in the *GCControlMap* object.

Figure 15 shows an example of a Trajectory object being mapped to the grain parameters controlling a three dimensional panning object used within the grain synthesis node. While only the pan parameter is specified in the cloud object, the GCTrajectoryMap object maps the Trajectory to the *panX, panY,* and *panZ* grain parameters.

```
// create a Trajectory and trajectory map object
traj = EnvTrajectory(
     Env([ 0, 10 ], [ 10 ], 'sine'),   // x-axis
     Env([ -10, 20 ], [ 10 ], 'sine'), // y-axis
     Env([ 0, 10 ], [ 10 ], 'sine')    // z-axis
);
obj = GCTrajectoryMap(traj);
```

Fig. 15. Code example of instantiating a GCTrajectoryMap used to map a trajectory signal to three parameter inputs.

GCBusMap

A GCBusMap object maps a grain parameter input to a control bus. It differs from the GCControlMap object in that the signal on the bus is not defined or controlled by the parameter object in any way. The signal on the bus could come from any source external to the cloud and parameter objects, as long as the synthesis node writing to the bus appears on the order-of-execution tree prior to the grain group node.

The only instance variable used by the GCBusMap object is the *bus* variable. The bus variable contains a SuperCollider Bus object representing the control bus to which the parameter input is mapped. The *controlMsg* method returns nothing, since the control node is handled externally. The *map* method returns the bus-mapping message that maps the parameter input to the specified control bus. Figure 16 shows an example of the use of the GCBusMap object.

```
// create a Bus object and map it
bus = Bus.control(server, 1);
obj = GCBusMap(bus);
```

Fig. 16. Code example of the instantiation of a GCBusMap object.

GCMidiMap

The GCMidiMap object maps a grain parameter input to values received from a MIDI controller. The object operates by setting up a MIDI responder in the client that sets the MIDI value on a control bus each time a message matching certain object criteria is received. The grain parameter input is then mapped to the control bus.

Values can be retrieved from *noteOn, noteOff, control, velocity, bend,* and *touch* MIDI event messages received from a specified MIDI source and channel. The event type, MIDI source, MIDI channel, and for certain event types a MIDI control value, can be set in instance variables within GCMidiMap messages named *type, src, channel,* and *value* respectively.

The *noteOn* and *nodeOff* event types map MIDI note values from a keyboard to the parameter input for noteOn or noteOff MIDI messages. The *control* event type maps the grain parameter to MIDI values of the controller number specified in the *value* parameter of the object. The *velocity* type maps the key velocity of a MIDI key specified in the *value* parameter to the parameter input. The *bend* and *touch* event types map MIDI pitch bend message and after-

touch values to parameters. Only the *control* and *velocity* type need a *value* set to identify the controller number or MIDI key number tracked.

The object also contains *mul* and *add* instance variables specifying a values will be multiplied and added to the MIDI value in order to scale and transpose the limited 0 to 127 MIDI value range to a range suitable for the parameter input. A *func* instance variable is available for advance users who may want to define their own custom MIDIResponder function used to capture values from MIDI event messages. If the *func* variable is not set, an appropriate responder function is used depending the *type* value.

The *prepare* method is important to the GCMidiMap object in order to build and set-up the MIDI responder that will parse the MIDI messages and to allocate a control bus to use. No value is returned to calls to the *controlMsg* or *next* methods. The *map* method will return the bus-mapping message needed for each grain. Figure 17 shows an example of the GCMidiMap object mapping a MIDI control number to a parameter input.

```
// instantiate object
obj = GCMidiMap( \control,          // MIDI Message type
    1,                      // MIDI Source Index
    1,                      // MIDI Channel Number
    7,                      // Control Number
    100, 20                 // multiplier and add values
);
```

Fig. 17. Code example of the instantiation of a GCMidiMap object based on a MIDI controller number 7 for channel number 1.

Extended GranCloud Objects

The objects discussed so far represent the core functionality of the GranCloud project. There are other related classes that I will not define in detail in this document, but are worth mentioning for the extended functionality they provide.

GranCloudGroup

It is common for a sound to be generated by overlapping multiple clouds with different characteristics. The GranCloudGroup object allows multiple cloud objects to be treated together as a synchronized group. The group object is instantiated, and multiple cloud objects are added to it. The group object then provides synchronous *prepare, play, pause,* and *stop* methods allowing users to call each method once on the group object rather than iterating through each cloud object to call the method manually on each.

The GranCloudGroup class also provides additional support for archiving objects to text files. The *writeArchive* and *readArchive* methods may be called on the object to save the group and all associated clouds to an archive using a single method call.

The group object has the ability to associate specific sound files with clouds and includes a method to automatically load the sound files into server buffers in preparation for granulation. This functionality was added to the group class rather than to individual cloud classes, so samples used by more than one

cloud only need to be loaded once into a shared buffer, rather than wasting memory by being loaded several times into individual buffers.

GranCloud*2Interface*

The GranCloud2Interface is a robust graphical interface for the building and editing of GranCloud2 and GranCloudGroup objects from a graphical user interface. The graphical interface allows the grain SynthDef and speaker configuration to be defined using a set of menu options. Control parameters may be added, and control data may be viewed and manipulated for each parameter in several different ways depending on the choice of parameter object type. A full discussion of the interface and its capabilities are beyond the scope of this document, so detailed information will not be included.

The interface is useful for composers to be able to easily visualize and manipulate the control data in parameter objects. Parameter data object values may be manipulated using sliders, envelope views, table views, or text. Users may modify data while playback is executing in order to set parameters based on aural cues instead of numeric or visual ones. The interface also has the ability to render synthesis to audio files for use in other applications, and it can read and write archives of the cloud data.

I used the interface in the composition of *Creo* to quickly set and manipulate cloud data based on aural feedback. Once the cloud or group objects were created and control data set, I archived the objects to text-files. The composition controller script could then read the archived objects to instantiate

them in memory as needed during the composition. Once instantiated, the speaker configuration and server parameters were set, and the clouds were ready for playback within the composition. Very little actual code was required in the composition controller to execute the synthesis, since all the control data was stored in the external archive. This cleaned up the code significantly, and it allowed large complex datasets for control data to be quickly and easily set, speeding up the compositional process significantly.

Chapter 4

Application of GranCloud in *Creo*

Granular synthesis was used in several different ways in *Creo* to generate sounds and effects, to create sounds and textures that seamlessly transform from one state to another, and to harmonize instrumental audio feeds. In this section I have illustrated specific examples of the use of GranCloud to generate granular synthesis in *Creo*.

Harmonization

Once common use for granular synthesis is altering the pitch level of an audio feed to harmonize live instruments. Using this technique, audio signals are written to a looping buffer that is used as the audio source for the grains. Since the buffer can be read at various speeds, the grain nodes can read the buffer faster or slower than the sample rate, resulting in a higher or lower sound. Many of these grains overlapping can create a continuous tone based on the original source timbre but transposed to a different pitch level.

If grain parameters are set correctly, granular harmonization is one of the better sounding methods of harmonization. It is, however, a very processor intensive method and may be unrealistic for harmonization in compositions that are already highly taxing the CPU.

Creo used harmonization in several places with varying degrees of CPU processing intensity. Granular harmonization with GranCloud was used in parts

of the composition where the CPU usage was low enough to accommodate it. In places where CPU usage was already high, harmonization was done using a built-in SuperCollider pitch shifting unit generator.

Figure 18 shows an example of GranCloud harmonization as used in *Creo.* In this application, a Phasor signal is used to coordinate the write position of the buffer with the read position of the grains, so that grains do not read over the write position. The buffer read position is calculated by subtracting the product of the grain duration, sample rate, and buffer read rate from the write phasor position. The grain duration is set to be dependant on the grain rate. The buffer read rate changes the pitch of the harmonization. A slight deviation is added to the grain rate to break up unnatural sounding periodicities created by perfectly synchronous grains.

Fig. 18. Example of harmonization in *Creo* using GranCloud (continues).

```
// Send grain SynthDef to server
SynthDef.new(\gc_harm_grain, {
    arg out, buf, dur, phasorBus, bufRate, amp, pan;
    var sound, bufPos;

    // Base buffer position on delayed Phasor signal
    bufPos = In.ar(phasorBus) -
        (dur * BufSampleRate.kr(buf) * bufRate);

    // Read sound from buffer
    sound = PlayBuf.ar(1, buf, bufRate, 1.0, bufPos,
        loop: 1);

    // Apply grain envelope
    sound = sound * EnvGen.kr(Env.sine(dur, amp),
        doneAction: 2);
```

```
        // Pan the output
        sound = Pan2.ar(sound, pan);

        // Write sound out to audio bus
        Out.ar(out, sound);

    }).send(server);

    // Define an audio rate bus for the Phasor
    phasorBus = Bus.audio(server, 1);

    // Allocate a buffer for the source sound
    buffer = Buffer.alloc(server, 44100, 1);

    // Instantiate GranCloud object and set parameters
    cloud = GranCloud2(server, \gc_harm_grain);
    cloud[\rate] = GCMinMax(0.01, 0.012);
    cloud[\dur]  = GCValue({ arg time, g; g[\rate] * 5 });
    cloud[\phasorBus] = GCValue(phasorBus.index);
    cloud[\bufRate]   = GCValue(1.5);
    cloud[\amp]  = GCValue(1);
    cloud[\pan]  = GCMinMax(-0.1, 0.1);
    cloud[\buf]  = GCValue(buffer.bufnum);

    // Start Phasor and record synth to write live signal
    // to the buffer. Include the possibility of a delay.
    delay = 0;
    synth = { var phasor, sound;
        // Read sound from audio interface input bus
        sound = AudioIn.ar(1, 1);

        // Delay input sound if the delay is set
        if(delay > 0, {
            sound = DelayN.ar(sound, delay, delay);
        });
```

(Fig. 18 continues)

```
        // Create phasor signal to control record position
        phasor = Phasor.ar(
                0,
                BufRateScale.kr(buffer.bufnum),
                0,
                BufFrames.kr(buffer.bufnum)
        );

        // Write audio signal to buffer at phasor position
        RecordBuf.ar(
                sound,
                buffer.bufnum,
                BufSampleRate.kr(buffer.bufnum)
        );

        // Write phasor signal to audio bus to be read by
        // grain synths
        Out.ar(phasorBus.index, phasor);
    }.play(server);


    // Prepare and Play the cloud
    cloud.prepare;
    cloud.play;
```

(Fig. 18 continued)

Generation of an Explosive Sound

About two minutes into *Creo,* there is an explosive sound symbolically representing the Big Bang and other concepts related to creation. I created the sound by mixing several granular clouds that are initially very dense and decrease in density over time. Low frequency clouds are derived from synthesized sine waves to create a deep rumble. I created higher frequency tinkling sounds by using a percussive envelope and a Blip unit generator. The Blip generator creates an audio signal that is the sum of a configurable number

of harmonic sine partials. The number of harmonics is randomized using a grain parameter to create a cloud derived from differing, but related, timbres. Eight clouds are also added to the mix that granulate various samples of breaking glass and ceramic items to create intense high frequency sounds.

I created all of these clouds using the GranCloud2Interface class. The clouds were created using the graphical interface and archived to text files that were stored with the composition controller. A few seconds before the clouds are needed in the composition, the controller reads the archives to instantiate the cloud objects in memory. It reads the sound files used into buffers, replaces the Server and SpeakerConfig objects in the cloud objects, calls the *prepare* method on the clouds. The *play* method is then executed at the appropriate time to generate the synthesis for the explosive sound.

The control data was defined in the cloud objects using GCValue objects containing static values or envelope objects depending on the parameter. The envelopes were mapped to graphical envelope or table views in the interface for easy manipulation based on visual and aural feedback. Every detail of the control data is too detailed to cover in depth in the scope of this paper. The grain SynthDefs and parameters that were used have been included in the figures below to demonstrate the structure of the synthesis.

Low Rumble Cloud

The grain SynthDef used for the low rumble is shown in figure 19. The pan position is controlled by a three-dimensional trajectory. Random motion is added

to each trajectory axis by adding random noise generators to each. The *panX, panY,* and *panZ* parameters are mapped to the Trajectory object using a GCTrajectoryMap to define the center position of the cloud as it moves through space. The *motion* parameter controls how fast the grains move on each axis, and the *diffusion* parameter controls how far from the cloud center grains are allowed to deviate on each axis.

A SinOsc unit generator creates the source sine wave. The frequency and amplitude of the oscillator is controlled by the *freq* and *amp* grain parameters. An Env object defines the grain envelope that is applied to the sine wave. It has a configurable curvature and peak location that are controlled by the *envCurve* and *envAttack* parameters. An EnvGen unit generator converts the Env to a signal of length *dur* that is applied to the sine wave. The EnvGen has a *doneAction* parameter equal to 2 indicating that the synthesis node should be freed from the server when the end of the envelope has been reached.

Finally the sound is mixed out between two audio busses with levels dependent on the value of the *mix* parameter. The *out* parameter indicates the audio bus to which the dry sound is written, and the *out2* parameter indicates an audio bus connected to a reverb processor.

```
SynthDef(\sine_grain, {
    arg out=0, out2=2, mix=0.4, rate=0.05, dur=0.1,
        amp=0.2, freq=440, envCurve=0, envAttack=0.01,
        panX=0, panY=1, panZ=0, motion=100,
        diffusion=25;
    var sound, panTraj;

    // Pan position comes from trajectory control node.
    // Random motion is applied to each axis.
    panTraj = Trajectory(panX, panY, panZ);

    panTraj = panTraj + Trajectory(
        LFNoise1.kr(motion, diffusion),
        LFNoise1.kr(motion, diffusion),
        LFNoise1.kr(motion, diffusion)
    );

    // Sound created from sine wave with variable
    // percussive envelope.
    sound = SinOsc.ar(freq, 0, amp);
    sound = sound * EnvGen.ar(
        Env.perc(envAttack, 1 - envAttack, 1, envCurve),
        1, amp, 0, dur, doneAction: 2
    );

    // Apply 3D pan based on speaker configuration.
    sound = ~config.panT(sound, panTraj);

    // Mix output to dry audio out and reverb busses.
    MixOut.ar(out, out2, sound, mix);
});
```

Fig. 19. Sine grain SynthDef used in *Creo* for low rumble.

Tinkling Blip Cloud

The grain SynthDef used for the cloud generating the tinkling sound is shown in figure 20. It is very similar to the Sine Grain example just given, except that the source sound is generated from a Blip unit generator. A *numHarmonics* parameter has been added to control the number of harmonic sine partials that Blip unit generator will sum to generate the source sound.

```
SynthDef(\blip_grain, {
    arg out=0, out2=2, mix=0.4, rate=0.05, dur=0.1,
        amp=0.2, numHarmonics=10, freq=440, envCurve=0,
        envAttack=0.01, panX=0, panY=1, panZ=0,
        motion=100, diffusion=25;
    var sound, panTraj;

    // Pan position comes from trajectory control node.
    panTraj = Trajectory(panX, panY, panZ);
    panTraj = panTraj + Trajectory(
        LFNoise1.kr(motion, diffusion),
        LFNoise1.kr(motion, diffusion),
        LFNoise1.kr(motion, diffusion)
    );

    // Sound is created from Blip ugen and a variable env
    sound = Blip.ar(freq, numHarmonics, amp);
    sound = sound * EnvGen.ar(
        Env.perc(envAttack, 1 - envAttack, 1, envCurve),
        1, amp, 0, dur, doneAction: 2
    );

    // Apply 3D pan based on speaker configuration.
    sound = ~config.panT(sound, panTraj);
    MixOut.ar(out, out2, sound, mix);
});
```

Fig. 20. Blip grain SynthDef used in *Creo* for tinkling sound.

Breaking Glass Clouds

The SynthDef used for the granulation of glass and ceramic sounds is shown in figure 21. Like the other examples, a Trajectory object controls the pan position; however, the grains defined by this SynthDef are not placed in random motion or diffused about the center position defined by the trajectory. All grains follow the trajectory path directly. This is in part because the grains for these clouds sound successively and are much longer than typical grains used in granular synthesis. The time delay between grains places each in a new location on the trajectory. In addition, the trajectories for each grain were carefully defined to completely surround the audience such that timing variations in the grain rate already create the effect of random motion throughout the entire performance space.

The sound sources for the breaking sounds are read from buffers using the PlayBuf unit generator. The *bufReadRate* parameter controls the perceived pitch of each grain by configuring how fast the signal is read from the buffer. The *bufPos* parameter controls where in the buffer the PlayBuf begins reading. The grain envelope has a configurable attack position, release position, and curvature controlled by the *envAttack, envRelease,* and *envCurve* parameters respectively.

```
SynthDef(\buf_grain, {
    arg out=0, out2=2, mix=0.4, rate=0.05, dur=0.1,
        amp=0.2, bufnum=0, bufReadPos=0.5,
        bufReadRate=1, envCurve=0, envRelease=0.2,
        envAttack=0.2, panX=0, panY=1, panZ=0;
  var sound, panTraj;

    // Pan controlled by a Trajectory without additional
    // random motion.
    panTraj = Trajectory(panX, panY, panZ);

    // The source sound is read from a buffer.
    sound = PlayBuf.ar(1, bufnum,
        bufReadRate * BufRateScale.kr(bufnum), 1,
        bufReadPos * BufFrames.kr(bufnum)
    );

    // The sound is enveloped by an Env with
    // configurable attack, release, and curvature.
    sound = sound * EnvGen.ar(
        Env([ 0, 1, 1, 0 ], [ envAttack,
            1 - envAttack - envRelease, envRelease ],
            envCurve
        ), 1, amp, 0, dur, doneAction: 2);

    // Apply 3D pan based on trajectory and
    // speaker configuration.
    sound = ~config.panT(sound, panTraj);

    // Mix output to dry audio out and reverb busses.
    MixOut.ar(out, out2, sound, mix);
}
```

Fig. 21. Buffer grain SynthDef used in *Creo* for breaking glass sounds.

GranCloud *Efficiency*

GranCloud was efficient enough to process all ten of these granular clouds simultaneously on a 2.0 GHz Dual CPU Macintosh G5 without issues or audio glitches. This is in addition to running the controller script logic, performing

frequency and amplitude analysis on four input signals, and running synthesis nodes for instrumental amplification and reverb processing.

Transformation of Water Sounds

The middle formal section of *Creo* involves a 190 second continuous transformation of water based sound generated by GranCloud. The computer music transforms a cloud from a chaotic mass of granular sound bits into realistic sounding ocean waves. The section represents the movement from chaos to order as random elements are gathered and organized in creative processes.

The transformation utilizes a single cloud object that granulates a 27 second stereo sample of ocean waves. The cloud object was built in the composition controller and control data assigned entirely using code, rather than using the GranCloud2Interface class to generate and manipulate the control data graphically. Figure 22 shows the code defining the cloud object and control data.

Fig. 22. GranCloud code example of water sound transformation (continues).

```
// SEND SYNTHDEF TO SERVER
SynthDef(\buf_grain, {
    arg out=0, out2=0, mix=0.5, dur=0.1, amp=0.2,
        bufReadRate=1, bufnum=0, bufReadPos=0.5,
        envAttack=0.5, panX=0, panY=1, panZ=0,
        diffusion=0, contour=1, bufReadRateScalar=1,
        ampScalar=1;
  var sound, traj;

    // Pan controlled by trajectory with random value
    // added to each axis depending on diffusion param.
  panX = panX + Rand(diffusion * -1, diffusion);
  panY = panY + Rand(diffusion * -1, diffusion);
  panZ = panZ + Rand(diffusion * -1, diffusion);
```

```
    traj = Trajectory.new(panX, panY, panZ);

      // Source sound reads from specified buffer at
      // bufReadRate and bufReadPos controlling the pitch
      // and starting position within the buffer.
    sound = PlayBuf.ar(1, bufnum, ( bufReadRate +
            ( bufReadRateScalar * contour) ) *
            BufRateScale.kr(bufnum), 1,
            bufReadPos * BufFrames.kr(bufnum)
      );

      // Quasi-Gaussian envelope with adjustable
      // peak position applied to source.
    sound = sound * EnvGen.ar(
            Env([0, 1, 0],[envAttack, 1 - envAttack],\sine),
            1, amp + (ampScalar * contour), 0, dur,
            doneAction: 2
      );


      // 3D panner based on trajectory and
      // speaker configuration.
    sound = config.panT(sound, traj);

      // Mix output to direct out and reverb busses
    MixOut.ar(out, out2, sound, mix);
}).send(server);

// LOAD SOUNDFILES TO BUFFERS ON SERVER.
wave_buf = Buffer.read(server,
    path ++ "/sounds/Waves/seawash_calm_exerpt1.aif.L");
wave_buf2 = Buffer.read(server,
    path ++ "/sounds/Waves/seawash_calm_exerpt1.aif.R");


// INSTANTIATE THE CLOUD OBJECT.
waves = GranCloud2.new(
    server,          // server object
    \buf_grain,      // SynthDef name
    190,             // cloud duration
    0,                     // playback delay
    outBus,          // main output audio bus
```

(Fig. 22 continues)

```
        reverbOutBus,  // reverb processor in bus
        targetGroup,     // target group for order-of-exec
        \addToTail       // order-of-exec relative to target
    );

    // SET GRAIN PARAMETER VALUES

    // Set grain rate (time between grain start times).
    waves[\rate] = GCCenterDev(
        Env([ 0.0015, 2.0 ], [ 190 ], \exp),  // center
        Env([ 0.001, 0.5 ], [ 190 ], \exp)    // deviation
    );

    // Define amount of grain overlap.
    waves[\overlap] = Env.new([4, 8], [ 190 ], \exp);

    // Set common param adjusting both amplitude and
    // bufReadRate simultaneously. Timing scrambled
    // to make each rendition each rendition unique.
    waves[\contour] = Env.new([ -0.2, 0, -0.8, 0.1, -0.6,
        0.2, -0.4, 0.3, -0.2, 0.4, 0, 0.5, -1 ],
        [ 10, 15, 20, 15, 20, 20, 15, 25, 10, 15, 15,
        10].scramble, \sine
    );

    // Set grain duration based on rate and overlap.
    waves[\dur] = { arg time, g; g[\rate] * g[\overlap] };

    // Set grain amplitude controlled by envelope.
    waves[\amp] = GCCenterDev(Env([0.1, 2.5, 1.0, 6.0 ],
        [ 0.25, 0.5, 190 ], \linear), 0.2);

    // Define how fast buffer is read, changing pitch.
    waves[\bufReadRate] = Env([ 0.25, 3.0, 2.0, 0.35 ],
        [ 0.25, 0.5, 190 ], \welch);

    // Randomize buffer choice between stereo channels.
    waves[\bufnum] = { [ wave_buf.bufnum, wave_buf2.bufnum
        ].choose };

    // Define start position for reading the buffer.
    waves[\bufReadPos] = GCCenterDev(0.5, 0.4);
```

(Fig. 22 continues)

```
// Change envelope from percussive to quasi-Gaussian.
waves[\envAttack] = Env([ 0.1, 0.5 ], [ 190 ], \exp);

// Control pan position by single point trajectory.
waves[\pan]       = GCTrajectoryMap(
        PointTrajectory.fromArray([ [ 0, 1, 0, 0 ] ])
);

// Control diffusion of grains away from cloud center.
waves[\diffusion] = Env([ 100, 1 ], [ 100 ], \welch);

// Scalar adjusting how much the contour parameter
// affects the bufReadRate.
waves[\bufReadRateScalar] = 0.5;

// Scalar adjusting how much contour parameter
// affects the grain amplitude.
waves[\ampScalar] = 2.0;

// PREPARE CLOUD, PASSING IN THE SPEAKER CONFIGURATION
waves.prepare(config);

// WAIT A SECOND FOR ASYNCRONOUS TASKS TO COMPLETE
1.0.wait;

// PLAY CLOUD STARTING THE SYNTHESIS ENGINE
cloud.play
```

(Fig. 22 continued)

The first step of the code example shows the grain SynthDef being defined and sent to the server. Grain localization is controlled using a Trajectory object defining the central position of the cloud in space. A random diffusion is added to each axis for each grain. The maximum limit of the diffusion is controlled by the *diffusion* parameter. The actual panning is performed using a trajectory panning method built into the custom spatialization classes used by *Creo.* The sound will be panned to different channels according to the

SpeakerConfig object stored in the global *config* variable. This allows the same cloud to be rendered on different speaker configurations without changing the control data.

The source sound is read from a buffer by the PlayBuf unit generator. The buffer number is configurable by grain, so each grain can read from a different buffer. The *bufReadRate* and *bufReadPos* parameters function in the same manner as explained in previous examples, except that the bufReadRate is also be altered by the multiplication of a *contour* parameter. The *contour* parameter allows a single value to affect both the *bufReadRate* and the amplitude of the grain simultaneously. The *bufReadRateScalar* and *ampScalar* variables are multiplied to the *contour* when applied, in order to specify how much the *contour* alters each parameter value.

The envelope is quasi-Gaussian in its default shape, but the *envAttack* parameter can be adjusted to move the peak forward, creating a more percussive envelope, or backward, creating a more ramp-like envelope. The enveloped sound is panned and written out to two audio busses according to the *mix* parameter in the same manner as the other examples.

At the start of the transformation, the grains are very tiny and very dense. An envelope defines a grain rate that starts at 0.0015 seconds between grain start times and ends at 2.0 seconds between start times. The grain duration is controlled by a function that is dependent on the grain rate multiplied by a grain overlap parameter, so as the grain density is decreased or the amount of overlap

is increased, so the duration is increased. An envelope controls the overlap parameter, starting with 4 grains of overlap and ending at 8. This means that the grain duration at the start of the cloud will be 0.006 seconds, and it will change to 16 seconds by the end of cloud. By slowly changing of the grain rate and duration the resulting sound transforms from a chaotic wind-like granular sound, to the recognizable sound of ocean waves surrounding the audience. Toward the end of the transformation, the grains are so long they are not perceived as granular anymore.

Once all the parameters are defined, the *prepare* method is called to send required control SynthDefs to the server and to create the grain group node that controls order-of-execution of the grains. Once prepared, the client thread waits for a second, and executes the *play* method on the cloud to generate the granular synthesis.

Chapter 5

Conclusion

GranCloud is a powerful application that greatly simplifies the tasks of defining and generating granular synthesis within SuperCollider. The flexibility, extendibility, and efficiency of the software were all very important to the composition of *Creo.* I needed a flexible application to fully customize the synthesis of many different types of clouds used by the composition. Extendibility was important to support the use and control of external software within the grain synthesis. A very efficient application was required to be able to generate multiple clouds with high grain densities simultaneously in real-time. The encapsulation of control data and abstraction of the synthesis engine within individual cloud objects simplified the construction of the controller script significantly.

Although it was programmed specifically to support *Creo,* GranCloud now stands alone as an elegant solution to many problems common to computer music using granular synthesis. These solutions allow composers to focus more on musical decisions and less on technological issues during the compositional process. It is my hope that this shift of focus will not only improve the musical quality of my own future works, but will inspire others using GranCloud to achieve greater musical heights in their own compositions.

Reference List

Adler, Samuel. *The Study of Orchestration.* 2nd ed. New York: W. W. Norton, 1989.

Barron, Frank. "Putting creativity to work." In *The Nature of Creativity: Contemporary Psychological Perspectives,* ed. Robert J. Sternberg, 76-98. New York: Cambridge University Press, 1988.

Camurri, Antonio. “Artificial Intelligence Architectures for Composition and Performance Environments.” In *Readings in Music and Artificial Intelligence,* ed. Eduardo R. Miranda, 163-188. Contemporary Music Studies, ed. Peter Nelson and Nigel Osborne. Amsterdam: Harwood Academic Publishers, 2000.

The Csound Book: Perspectives in Software Synthesis, Sound Design, Signal Processing, and Programming, ed. Richard Boulanger. Cambridge, Massachusetts: The MIT Press, 2000.

Dannenberg, Roger. “Dynamic Programming for Interactive Music Systems.” In *Readings in Music and Artificial Intelligence,* ed. Eduardo R. Miranda, 189-206. Contemporary Music Studies, ed. Peter Nelson and Nigel Osborne. Amsterdam: Harwood Academic Publishers, 2000: 189-206.

Dean, Roger T. *Hyperimprovisation: Computer-interactive Sound Improvisation.* Computer Music and Digital Audio Series, v. 19. Middleton, Wisconson: A-R Editions, 2003.

Dick, Robert. *The Other Flute: A Performance Manual of Contemporary Techniques.* New York: Oxford University Press, 1975.

Forte, Allen. *The Structure of Atonal Music,* (New Haven and London: Yale University Press, 1973).

Kramer, Jonathan D. *The Time of Music: New Meanings, New Temporalities, New Listening Strategies.* New York: Schirmer Books, 1988.

Lee, Terry A. “GranCloud: A new SuperCollider Class for Real-time Granular Synthesis.” In *Multidimensionality: Proceedings of the International Computer Music Conference Held in New Orleans 6-11 November 2006,* edited by Suvisoft Oy Ltd., 55-62. San Francisco: The International Computer Music Association and New Orleans: Tulane University, 2006.

Morwood, James, ed. *The Pocket Oxford Latin Dictionary (Latin-English).* Oxford: Oxford University Press, 1994.

Roads, Curtis. *The Computer Music Tutorial.* Cambridge, Massachusetts: The MIT Press, 1994.

Roads, Curtis. *Microsound.* Cambridge, Massachusetts: The MIT Press, 2001.

Rowe, Robert. “Interactive Music Systems in Ensemble Performance.” In *Readings in Music and Artificial Intelligence,* ed. Eduardo R. Miranda, 145-162. Contemporary Music Studies, ed. Peter Nelson and Nigel Osborne. Amsterdam: Harwood Academic Publishers, 2000: 145-162.

Rowe, Robert. *Interactive Music Systems: Machine Listening and Composing*. Cambridge, Massachusetts: The MIT Press, 1993.

Rowe, Robert. *Machine Musicianship*. Cambridge, Massachusetts: The MIT Press, 2001.

Stone, Kurt. *Music Notation in the Twentieth Century: A Practical Guidebook.* 1st ed. New York: W. W. Norton, 1980.

Winkler, Todd. *Composing Interactive Music: Techniques and Ideas Using Max.* Cambridge, Massachusetts: The MIT Press, 2001.

PART II

CREO SCORE

creo

for
Flute, Violin, French Horn, Piano,
and
Live Interactive Electronics

by

Terry A. Lee

2009

PROGRAM NOTES

Creo is a Latin word meaning: “to create, make.” The composition explores the concept of creation from several different, and sometimes opposing, viewpoints. The music contains allusions to universal scientific theory, to religious views on creation, as well as to the individual and collaborative processes people experience when involved in the act of creating. The listener is invited to ponder how tidbits of information gathered from life experiences coalesce into ideas that eventually become physical inventions or works of art; how chaotic particulate matter spewed forth from the Big Bang combines to form stars, planets, and eventually a primordial soup from which life emerges; how God moves upon the face of the deep, separating the waters, setting the times and seasons, and creating man in His own image.

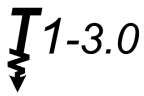
In the beginning…	conception	…a Big Bang!
Divide the waters…	organization	…Primoridal Soup.
Let the dry land appear…	aggregation	…Pangaea.
Let the earth bring forth…	assembly	…Origin of Species.
In our own image...	refinement	…Descent of Man.
The seventh day…	completion	…Theory of Everything.

PERFORMANCE NOTES

Spatial Notation: Sections that are notated spatially have a meter signature with a digit directly over an X. The digit indicates how many conductor queues will be given within the measure. The time between the start of the measure and the next bar line is indicated with a “ca.” and a number above the staff. Notes within the measure should be played in time according to their relative position within the measure. Notes should be sustained for the relative duration of the trailing flag attached to the notes. A vertical dash will indicate notes that should be played simultaneously between two or more parts. Accidentals within spatially notated sections apply only to the note directly following the accidental and to any notes directly repeated following a note with an accidental.

Traditional Notation: Traditionally notated sections have a normal meter signature and follow all rules of traditional notation. Accidentals within metered section follow traditional rules and carry over through the bar.

Symbols with an italicized *P,* downward arrow, and a number indicate trigger notes that the computer will be listening for to coordinate interaction with the instruments. The *P* indicates this is a pitch-based trigger and the number indicates a corresponding trigger number in the software. The computer will listen for the frequency surrounding the note directly under the arrow before proceeding, so it is important that these pitches are played clearly and accurately, or the computer may miss the trigger.



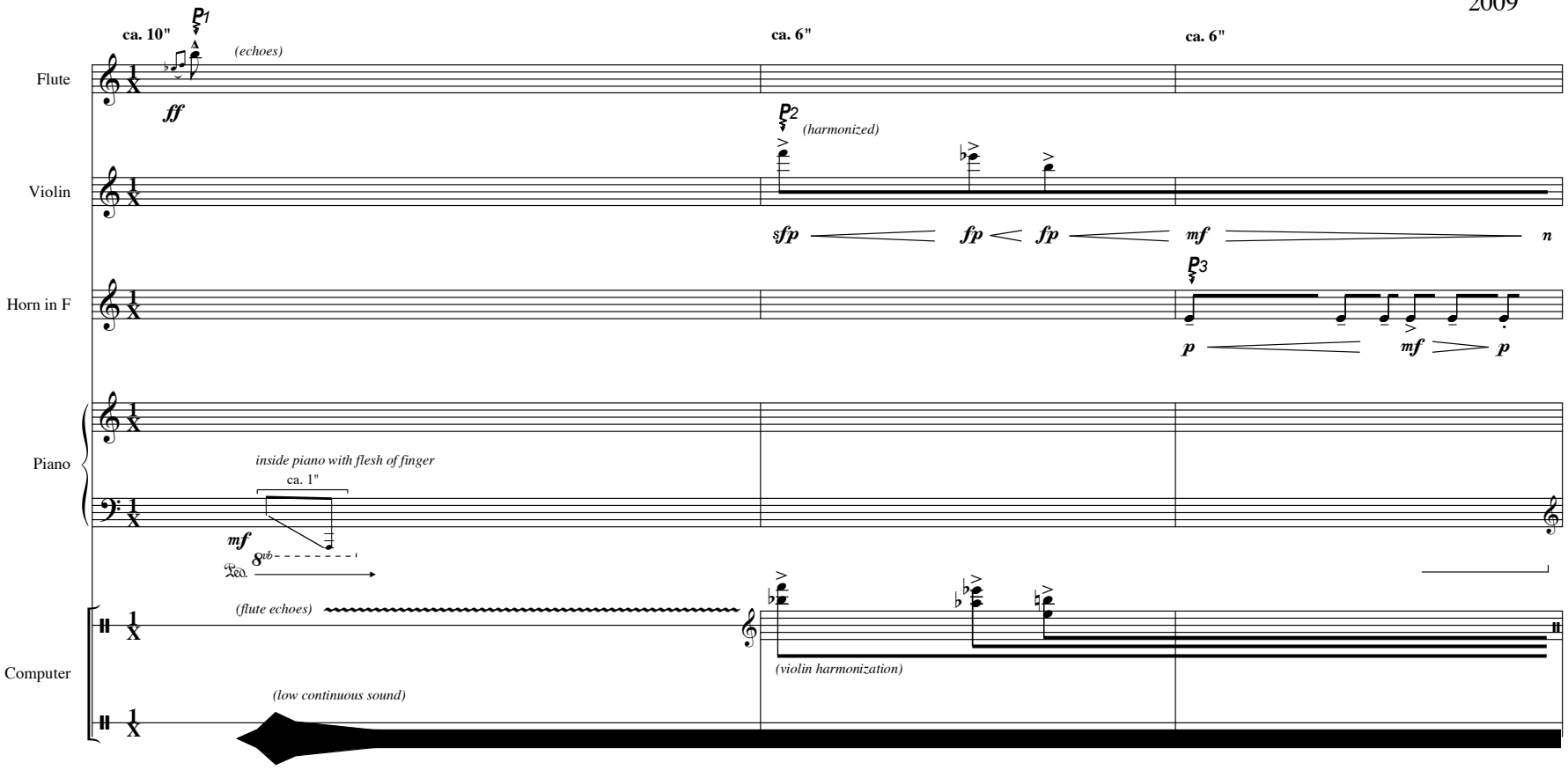
Symbols with an italicized T, downward arrow, and two numbers separated by a dash indicate a timed trigger point. The first number indicates the corresponding trigger number in the software. The second number indicates the amount of time in seconds that the computer will wait after the previous trigger was encountered before executing the next triggered event. Since these triggers are executed automatically, performers need not be concerned with them except to coordinate their own entrances with sound events executed by these triggers.

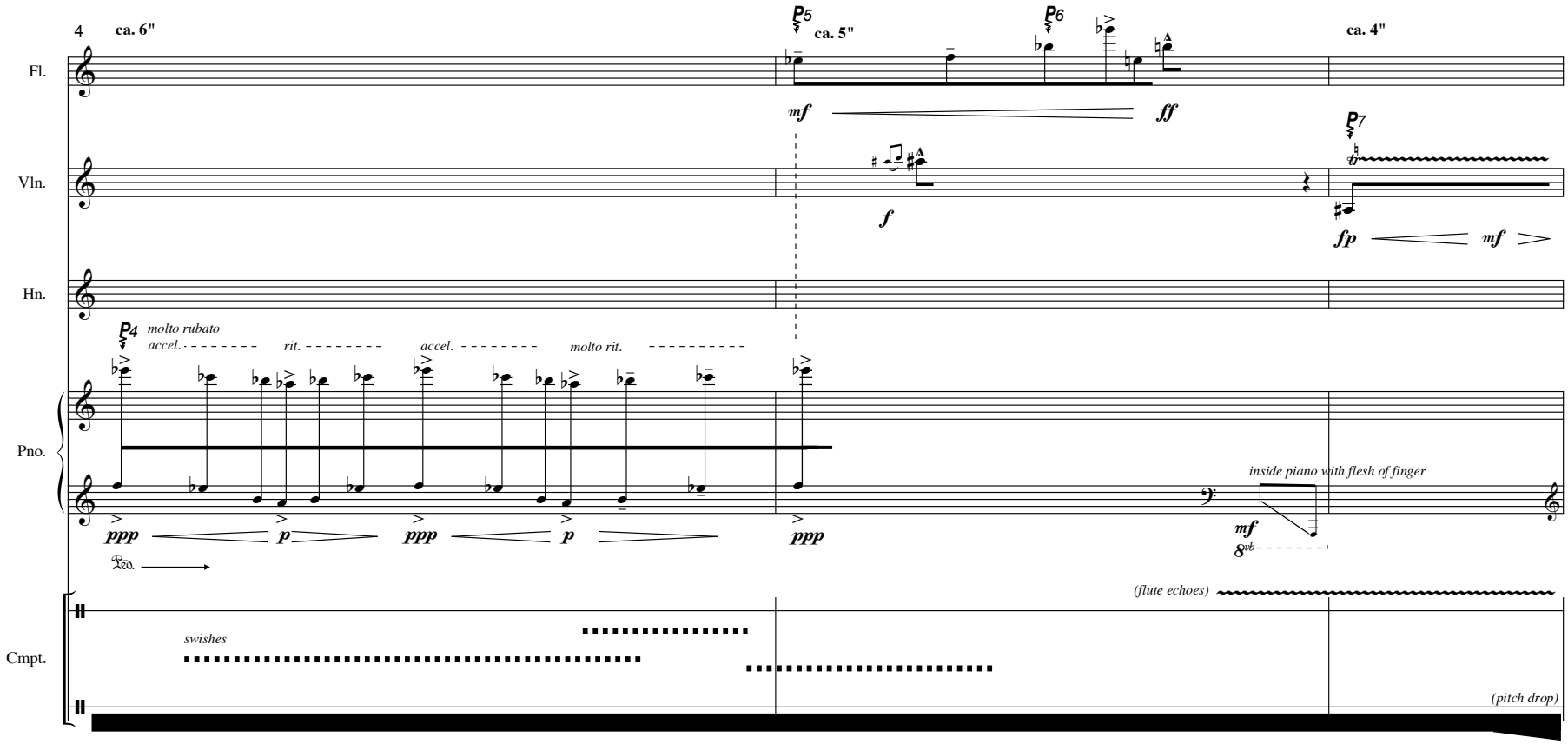
creo

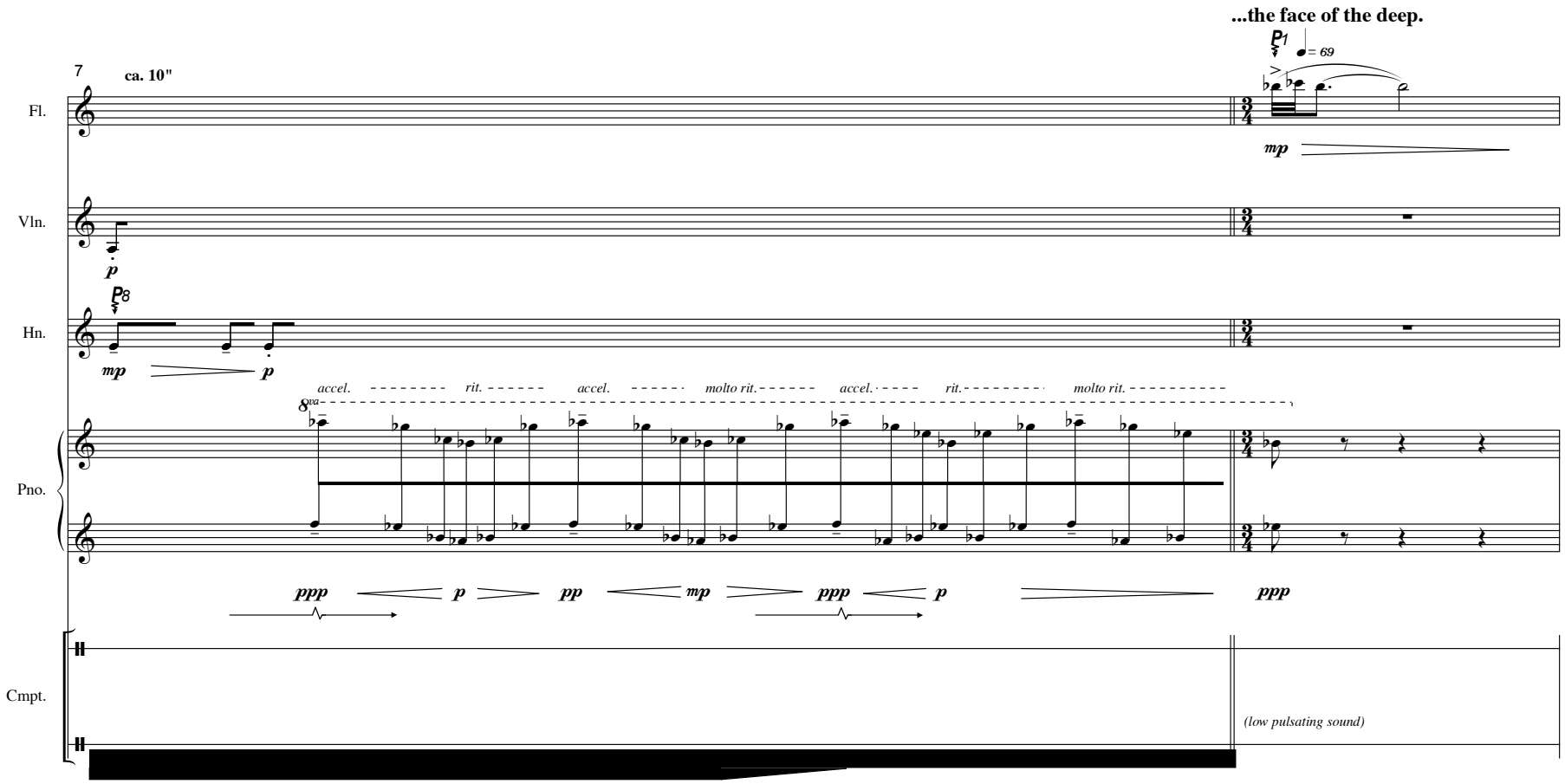
(transposed score)

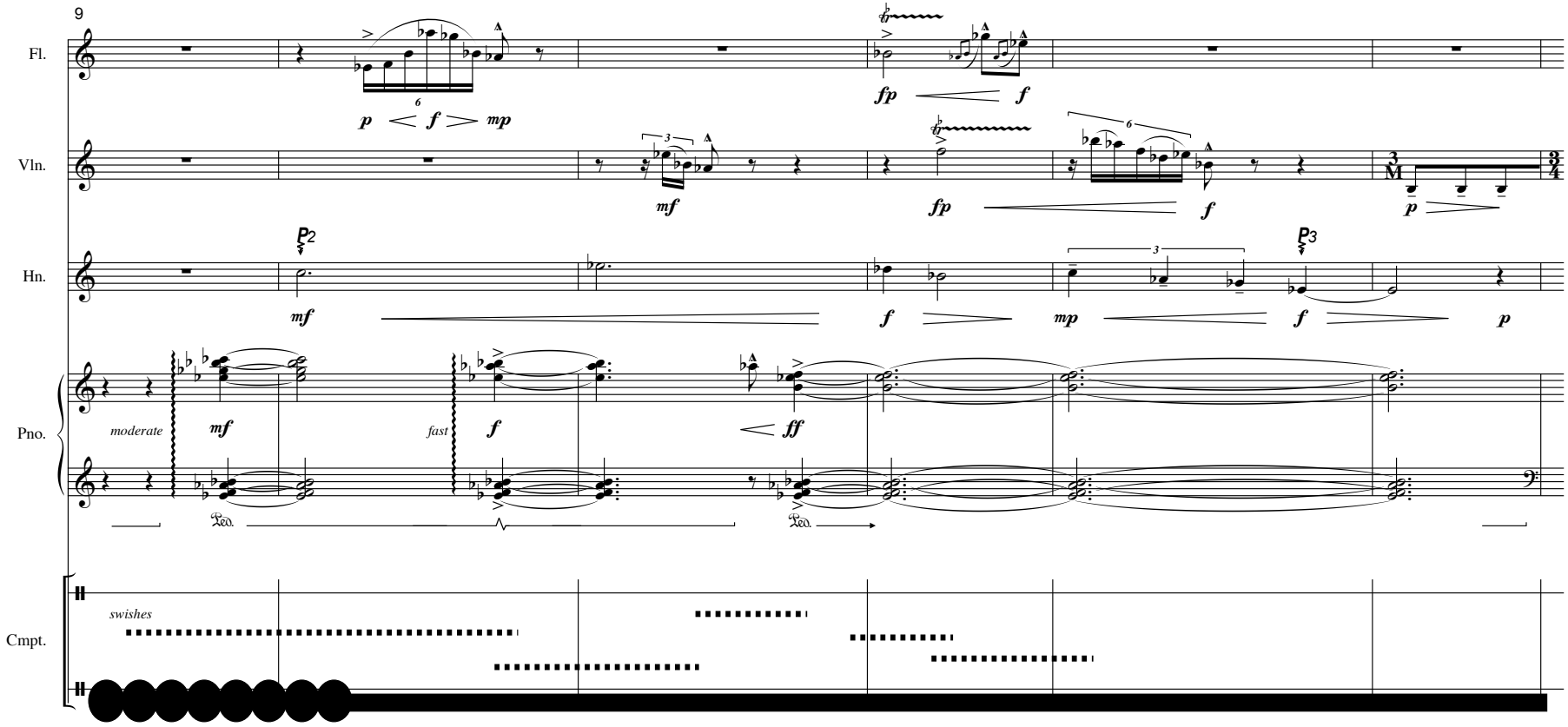
Terry A. Lee
2009

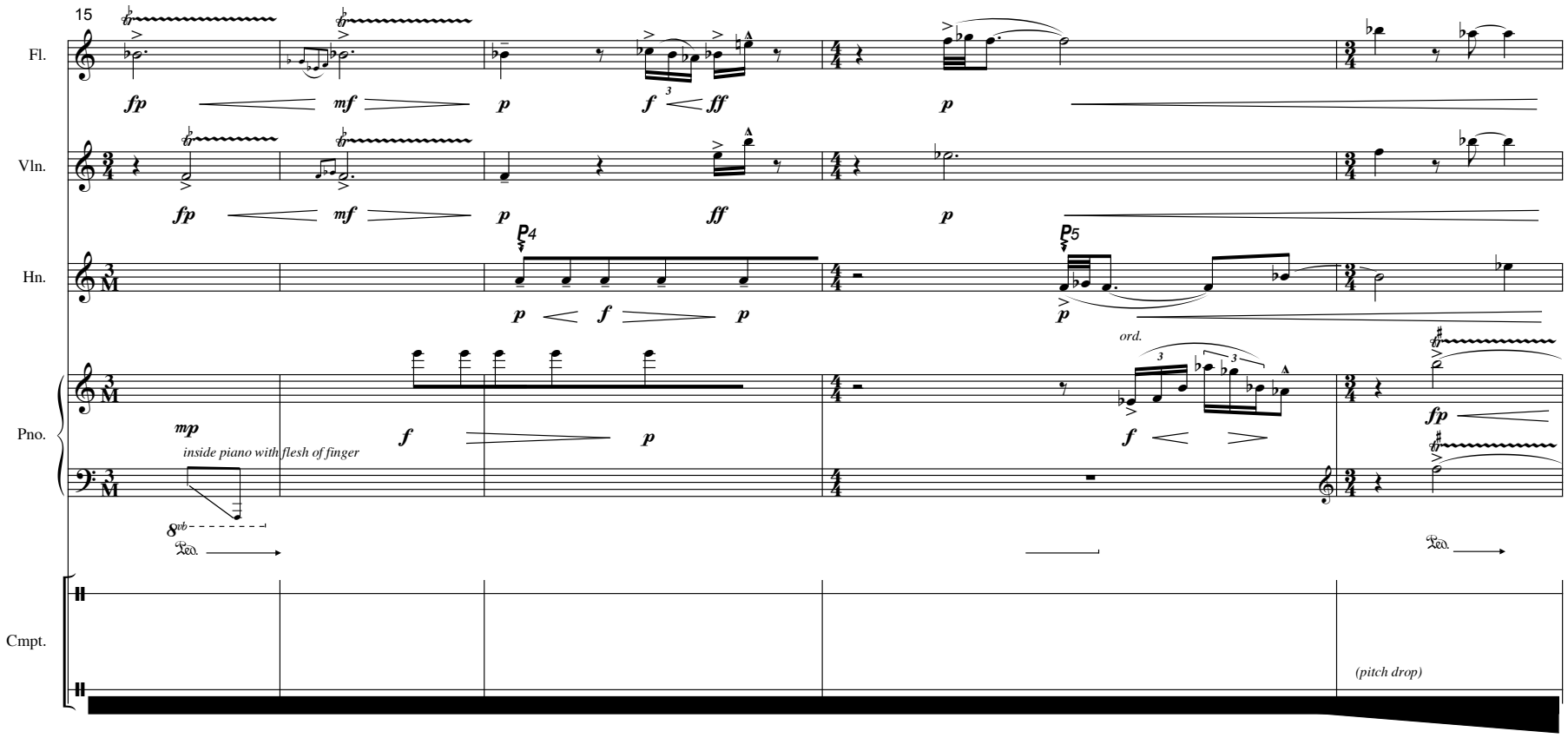
In the beginning...

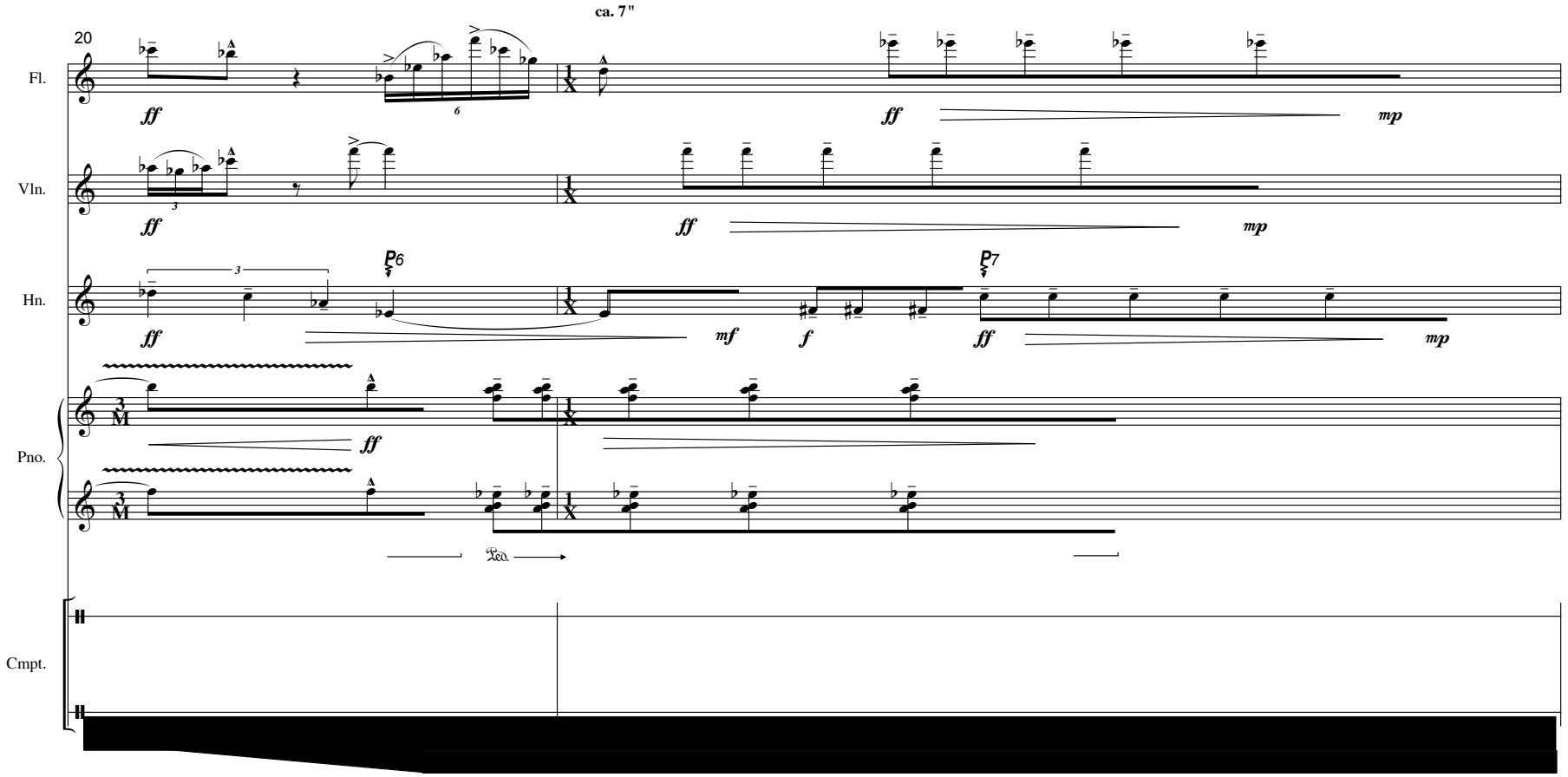


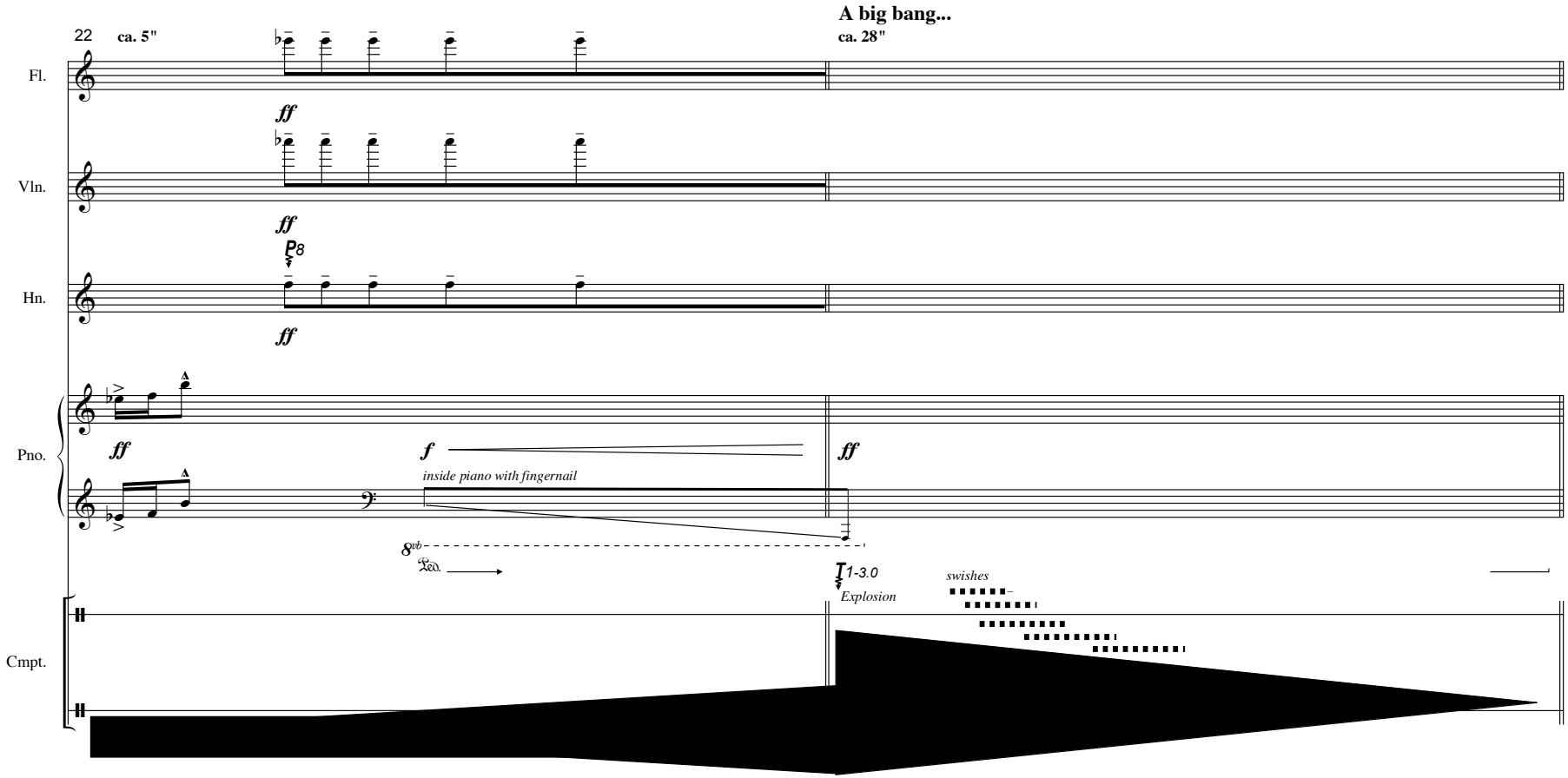

4
ca. 6"
P5
ca. 5"
P6
ca. 4"
Fl.
mf
ff
Vln.
P7
f
fp
mf
Hn.
P4
molto rubato
accel.
rit.
accel.
molto rit.
Pno.
ppp
p
ppp
p
ppp
inside piano with flesh of finger
mf
8vb
(flute echoes)
Cmpt.
swishes
(pitch drop)

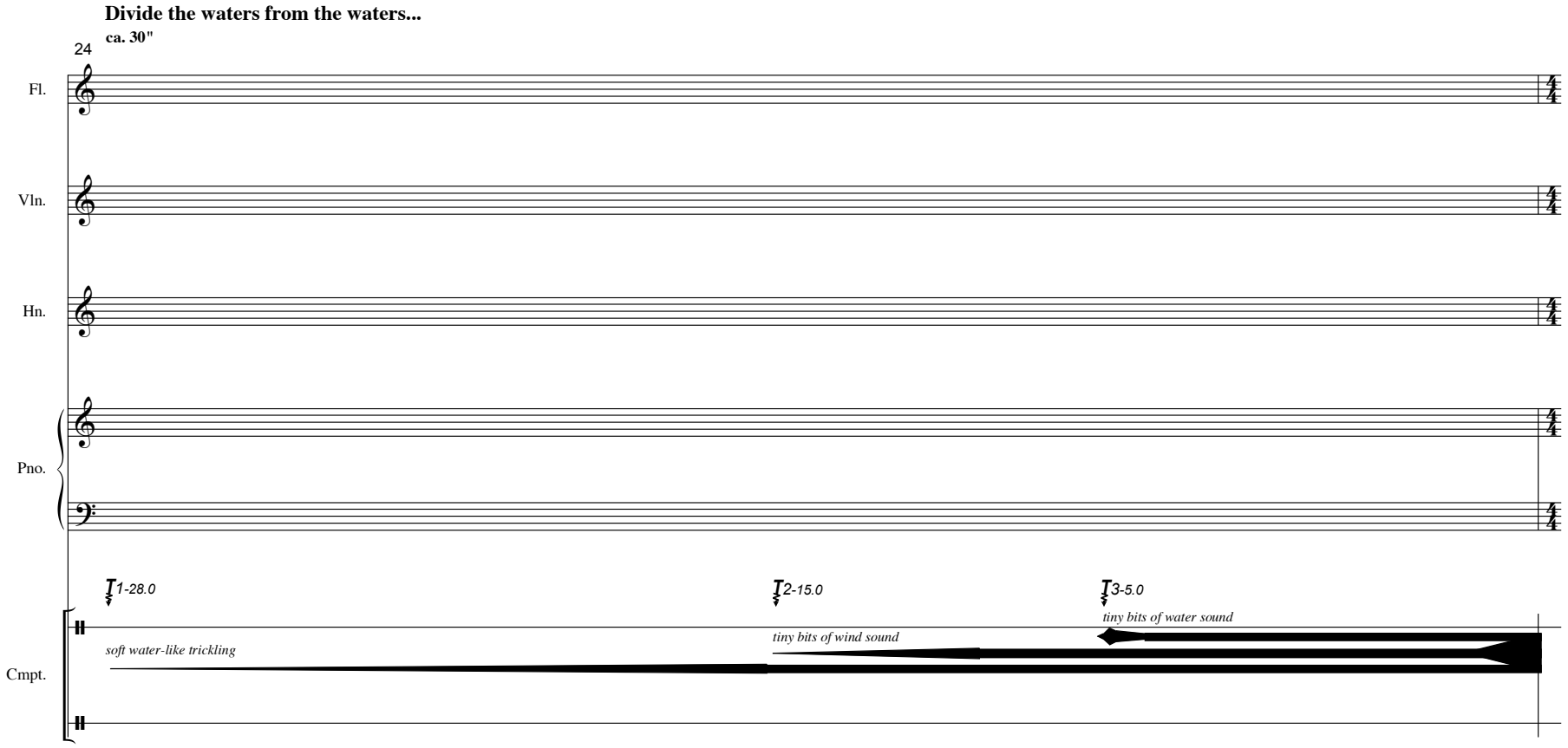

...the face of the deep.
7
ca. 10"
Fl.
Vln.
Hn.
Pno.
Cmpt.
= 69
mp
p
mp
p
accel.
rit.
accel.
molto rit.
accel.
rit.
molto rit.
8va
ppp
p
pp
mp
ppp
p
ppp
(low pulsating sound)

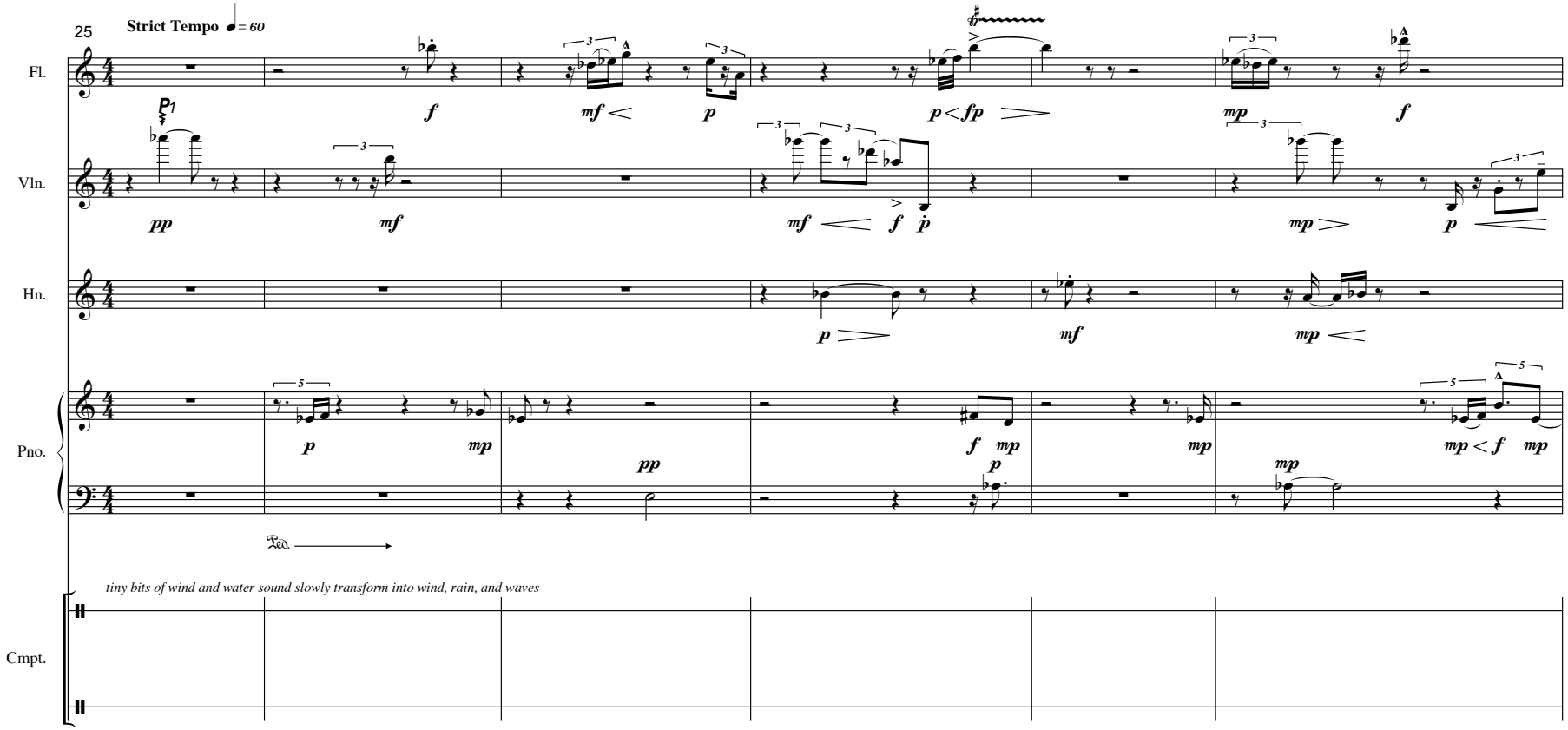

9
Fl.
Vln.
Hn.
Pno.
Cmpt.
p
f
mp
fp
mf
P2
P3
moderate
fast
ff
swishes

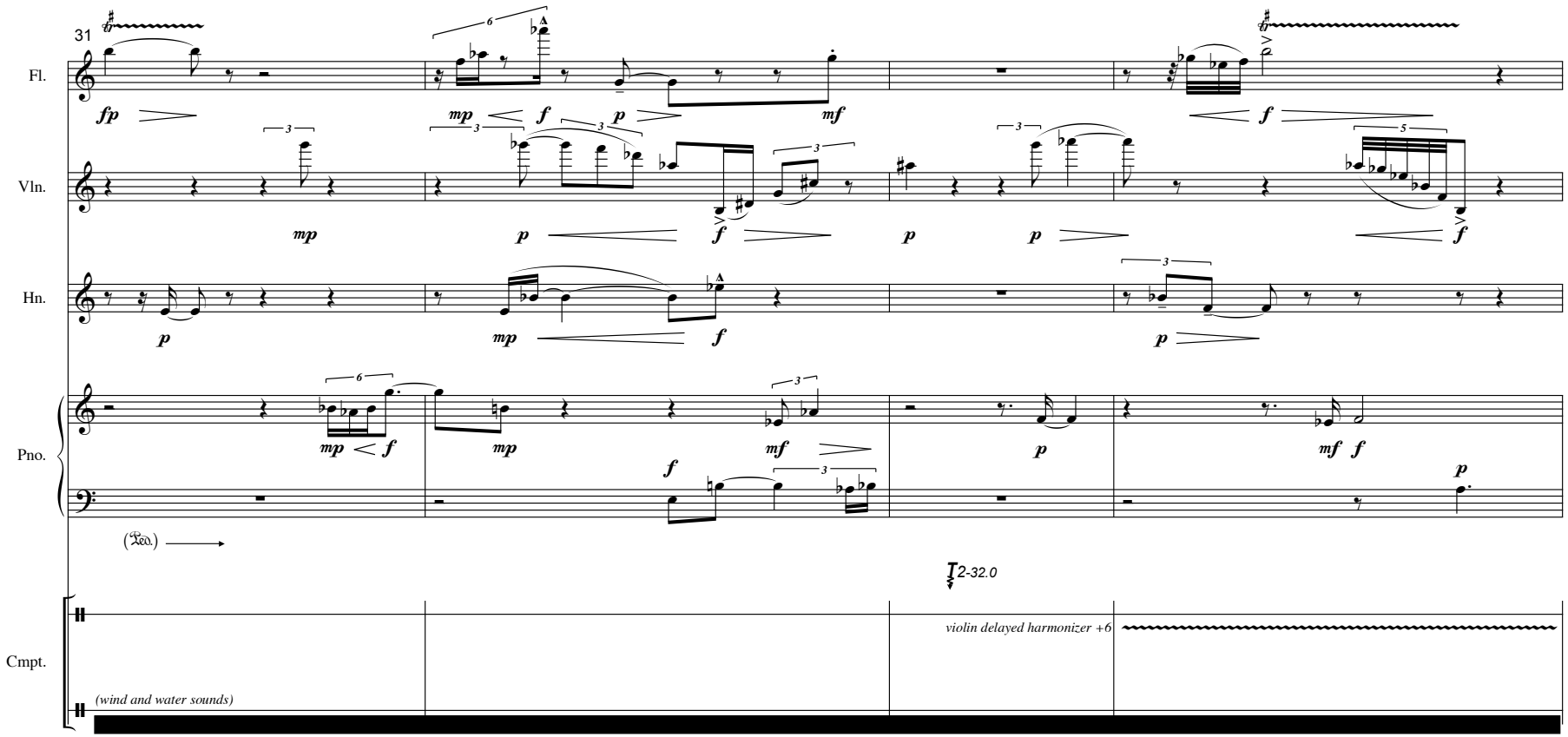

15
Fl.
Vln.
Hn.
Pno.
Cmpt.
fp
mf
p
f
ff
P4
P5
mp
inside piano with flesh of finger
ord.
(pitch drop)

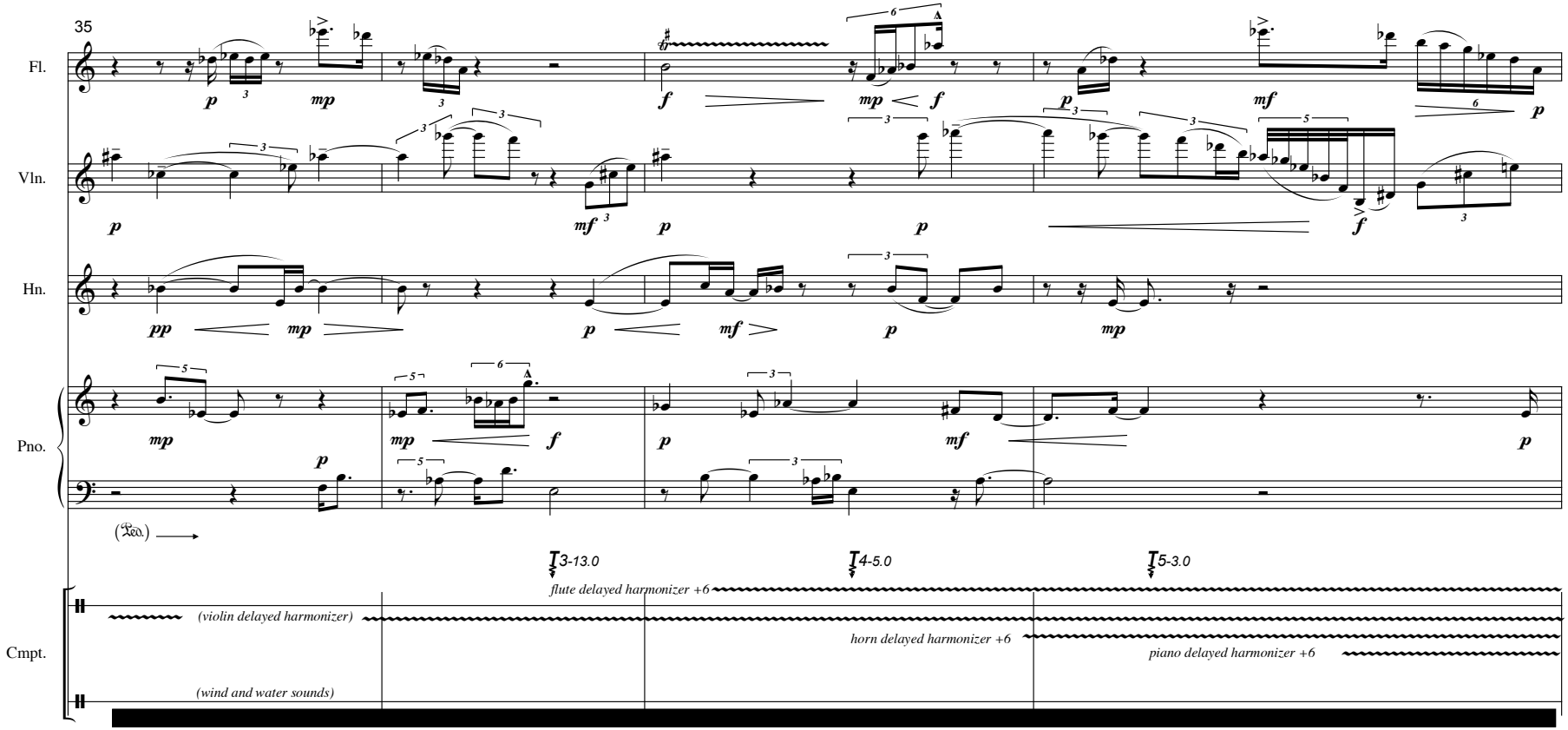

ca. 7"
20
Fl.
Vln.
Hn.
Pno.
Cmpt.
ff
mp
mf
f
P6
P7

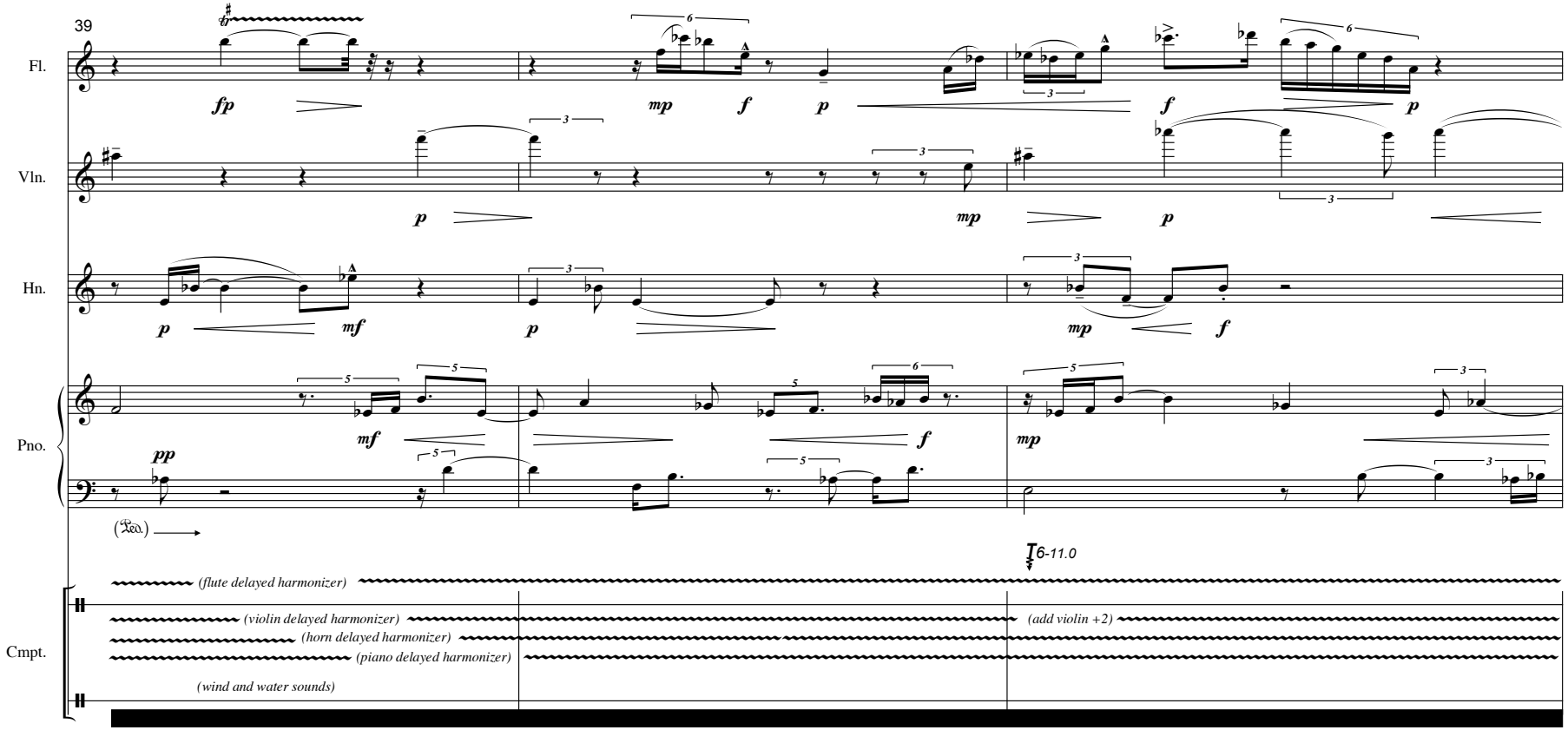

A big bang...
22
ca. 5"
ca. 28"
Fl.
Vln.
Hn.
Pno.
Cmpt.
ff
P8
f
inside piano with fingernail
8vb
Ped.
T1-3.0
Explosion
swishes

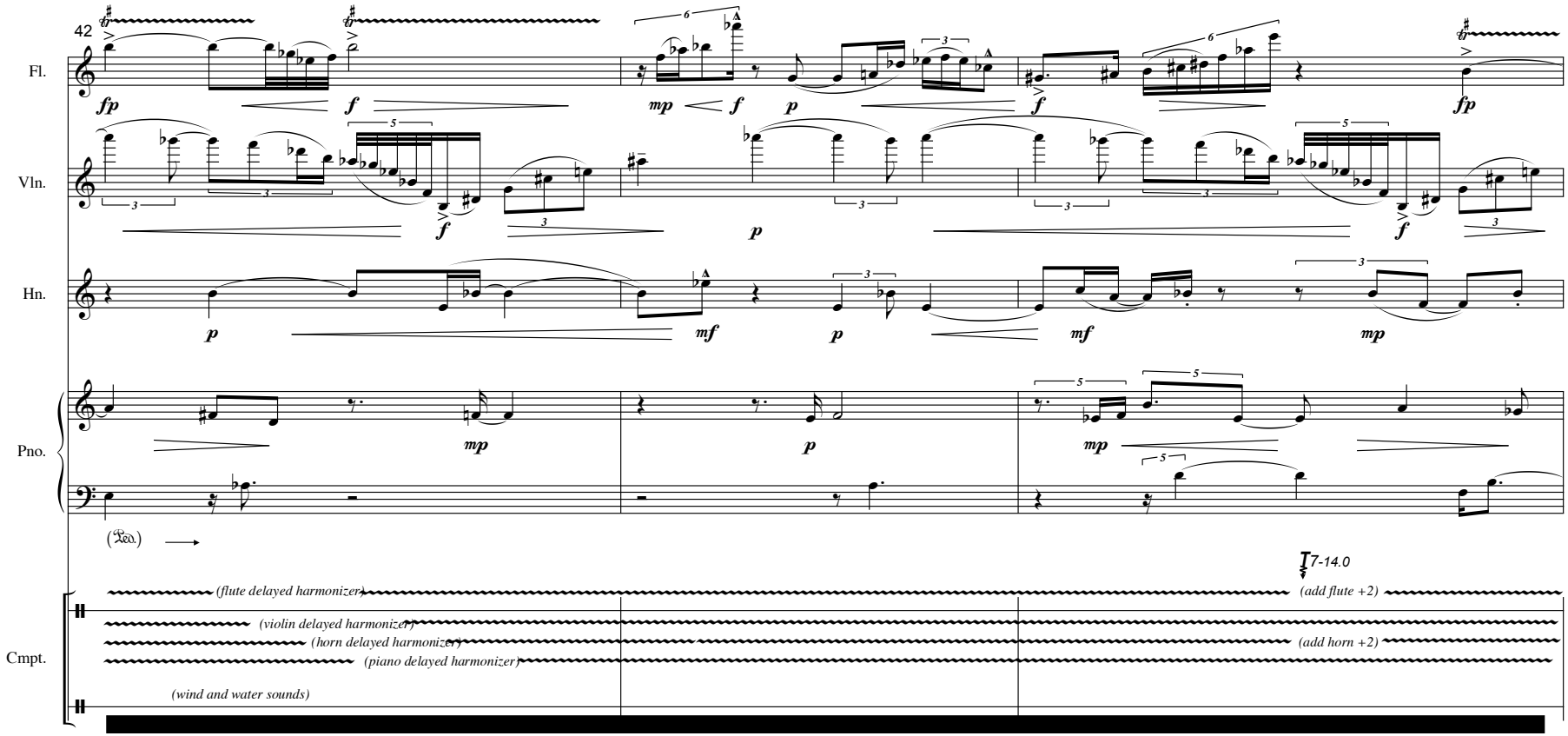

Divide the waters from the waters...
ca. 30"
24
Fl.
Vln.
Hn.
Pno.
Cmpt.
1-28.0
soft water-like trickling
2-15.0
tiny bits of wind sound
3-5.0
tiny bits of water sound

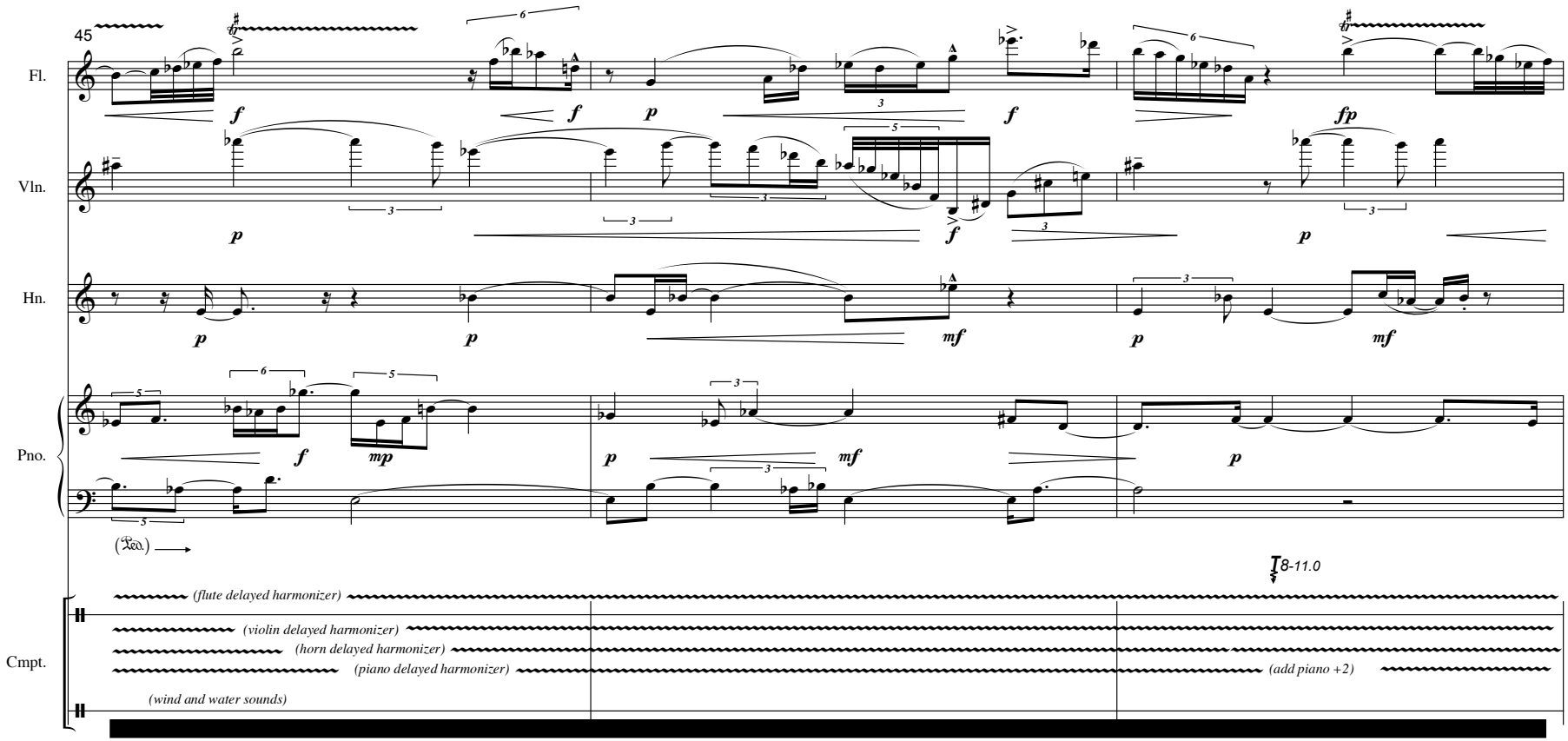

25
Strict Tempo ♩= 60
Fl.
Vln.
Hn.
Pno.
Cmpt.
P1
tiny bits of wind and water sound slowly transform into wind, rain, and waves

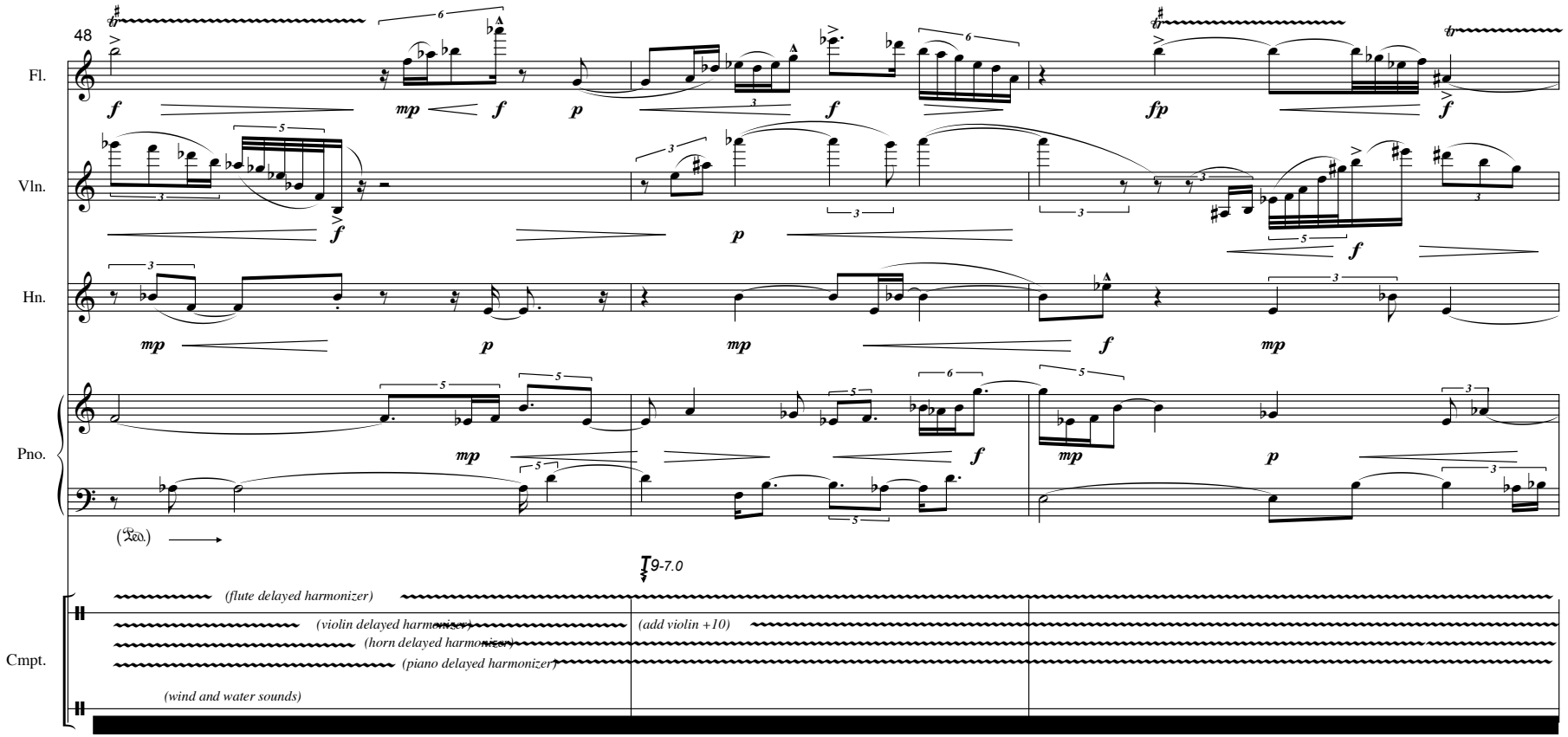

31
Fl.
Vln.
Hn.
Pno.
Cmpt.
fp
mp
f
p
mf
(wind and water sounds)
2-32.0
violin delayed harmonizer +6

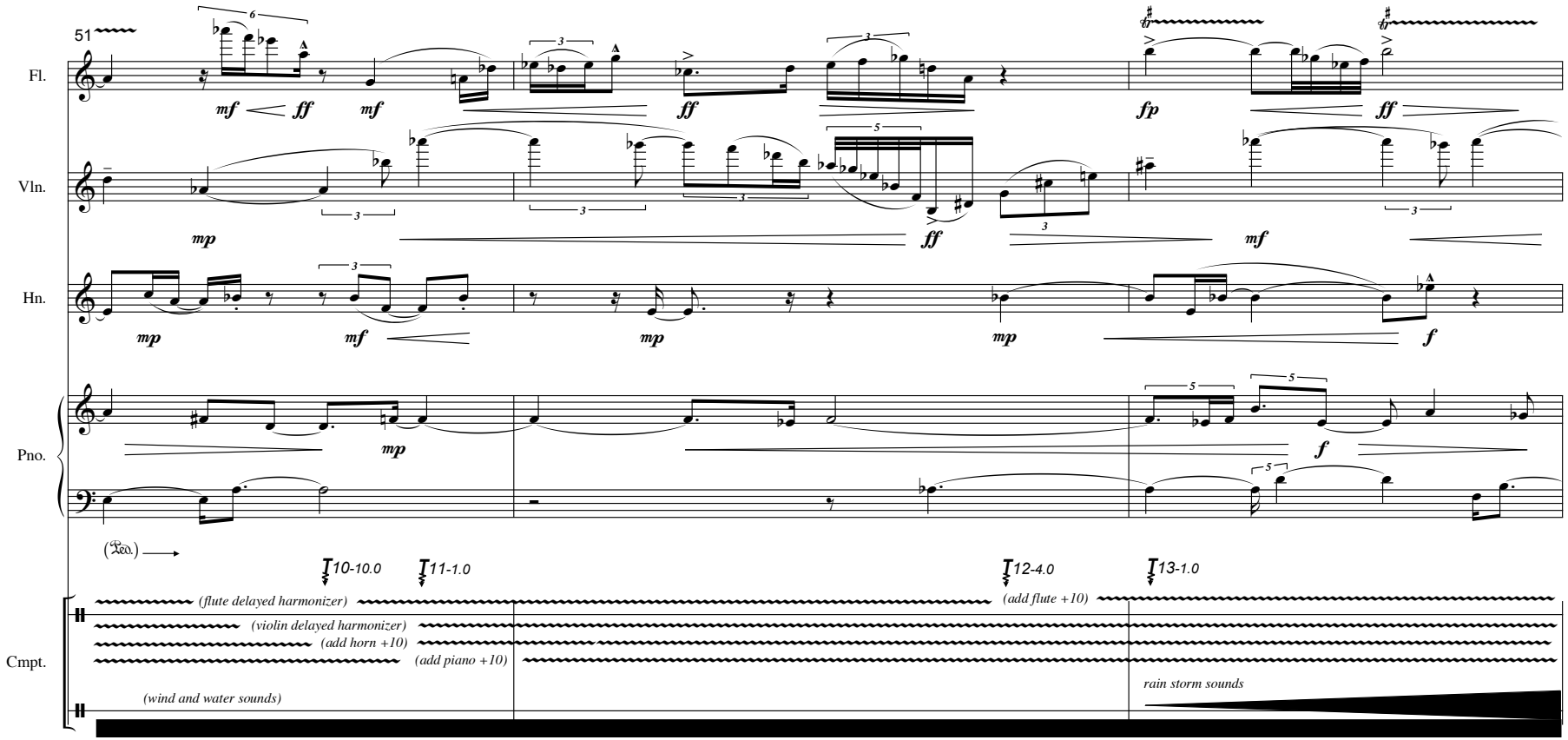

35
Fl.
Vln.
Hn.
Pno.
Cmpt.
(violin delayed harmonizer)
(wind and water sounds)
T3-13.0
flute delayed harmonizer +6
T4-5.0
horn delayed harmonizer +6
T5-3.0
piano delayed harmonizer +6

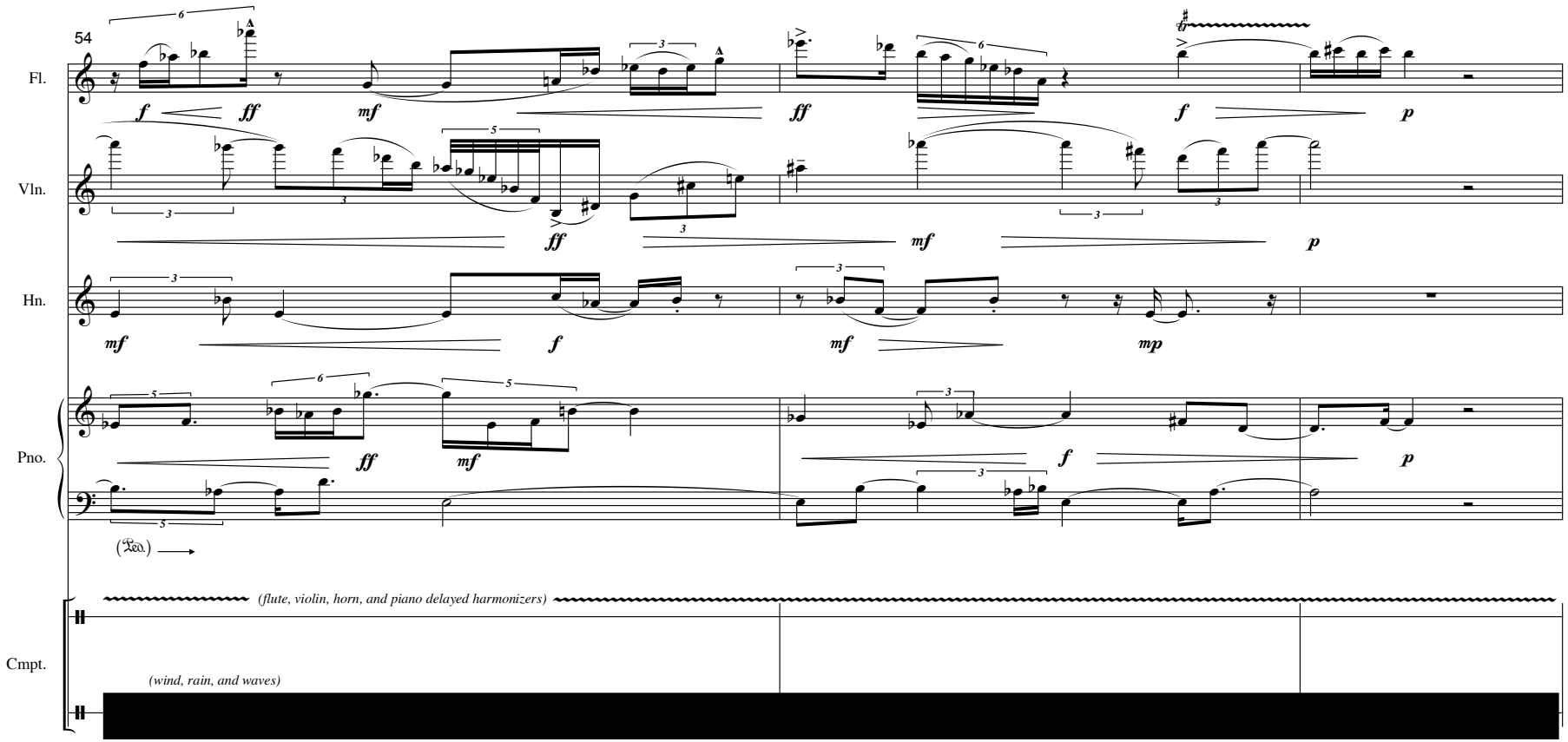

39
Fl.
Vln.
Hn.
Pno.
Cmpt.
fp
mp
f
p
mf
pp
(flute delayed harmonizer)
(violin delayed harmonizer)
(horn delayed harmonizer)
(piano delayed harmonizer)
(wind and water sounds)
T6-11.0
(add violin +2)

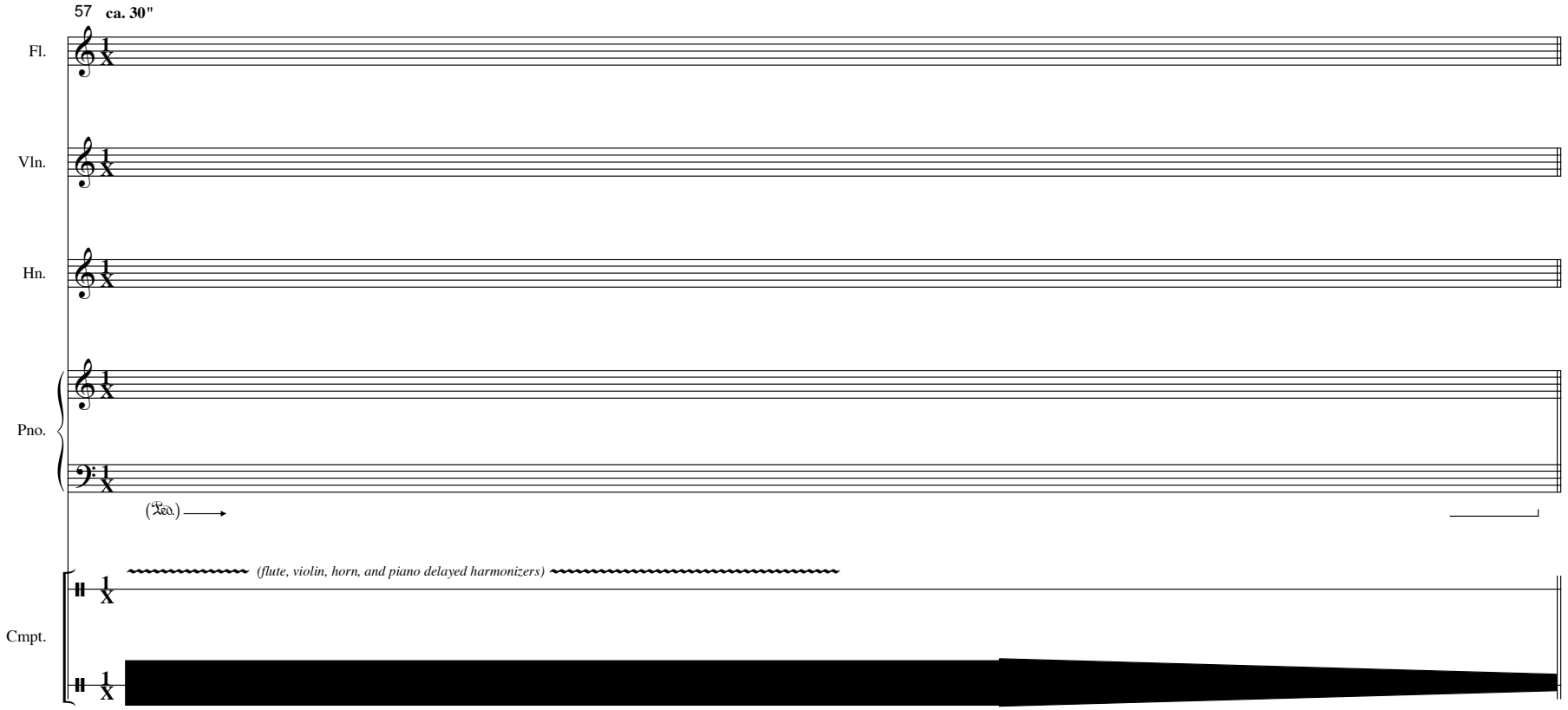

42
Fl.
Vln.
Hn.
Pno.
Cmpt.
fp
f
mp
p
mf
(Ped.)
7-14.0
(flute delayed harmonizer)
(violin delayed harmonizer)
(horn delayed harmonizer)
(piano delayed harmonizer)
(add flute +2)
(add horn +2)
(wind and water sounds)

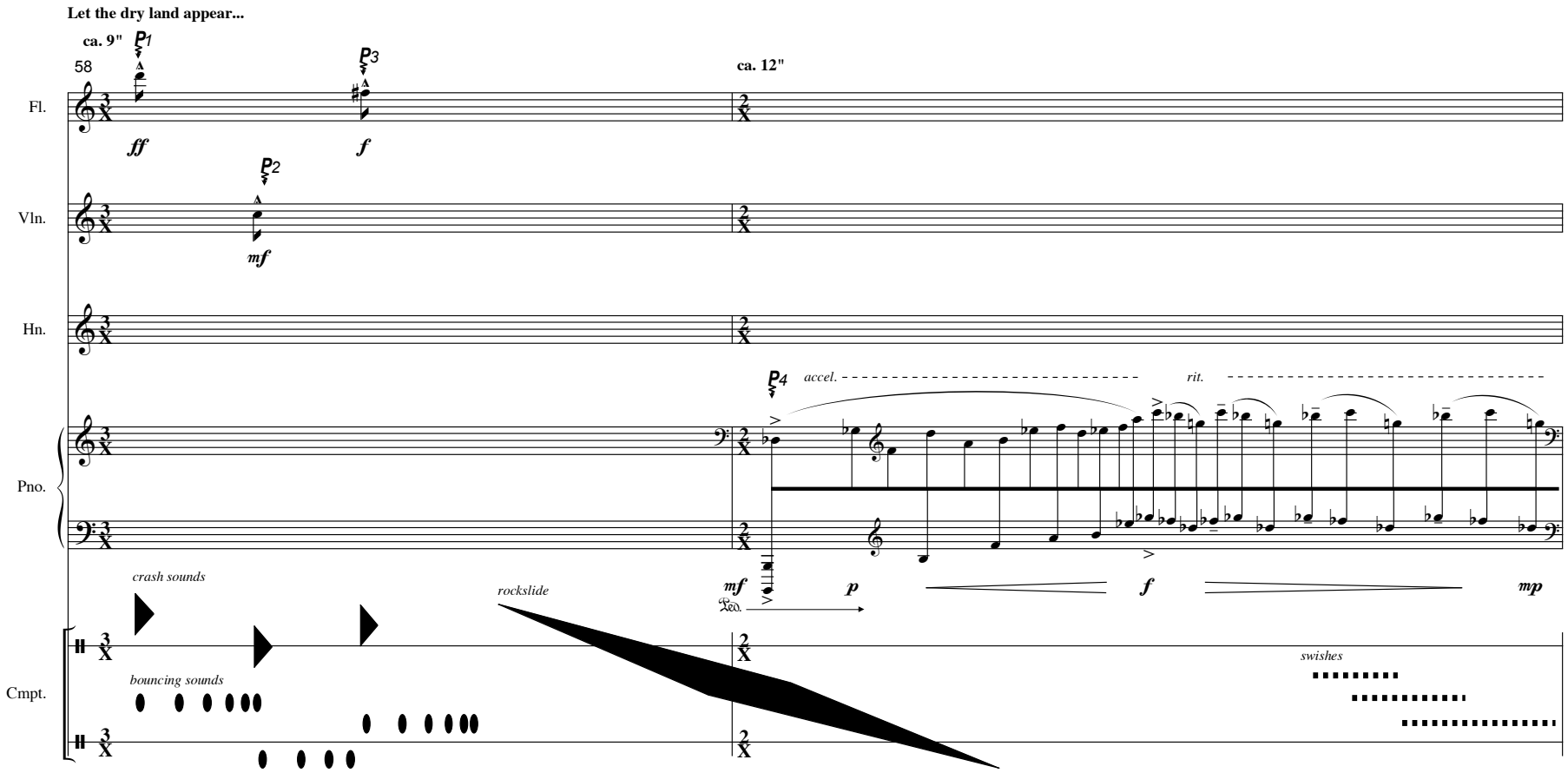

45
Fl.
Vln.
Hn.
Pno.
Cmpt.
(flute delayed harmonizer)
(violin delayed harmonizer)
(horn delayed harmonizer)
(piano delayed harmonizer)
(wind and water sounds)
8-11.0
(add piano +2)

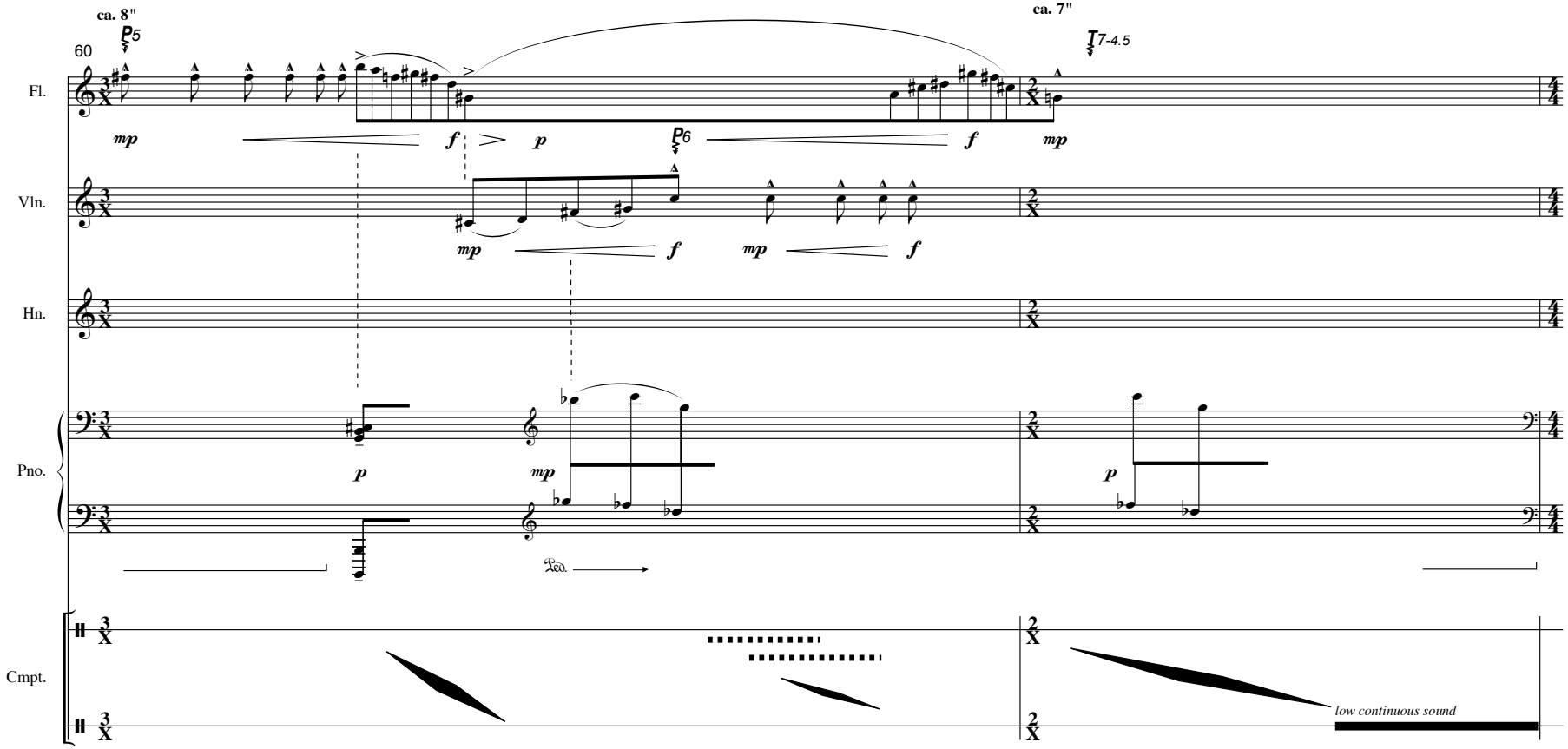

48
Fl.
Vln.
Hn.
Pno.
Cmpt.
(flute delayed harmonizer)
(violin delayed harmonizer)
(horn delayed harmonizer)
(piano delayed harmonizer)
(wind and water sounds)
(add violin +10)
T 9-7.0

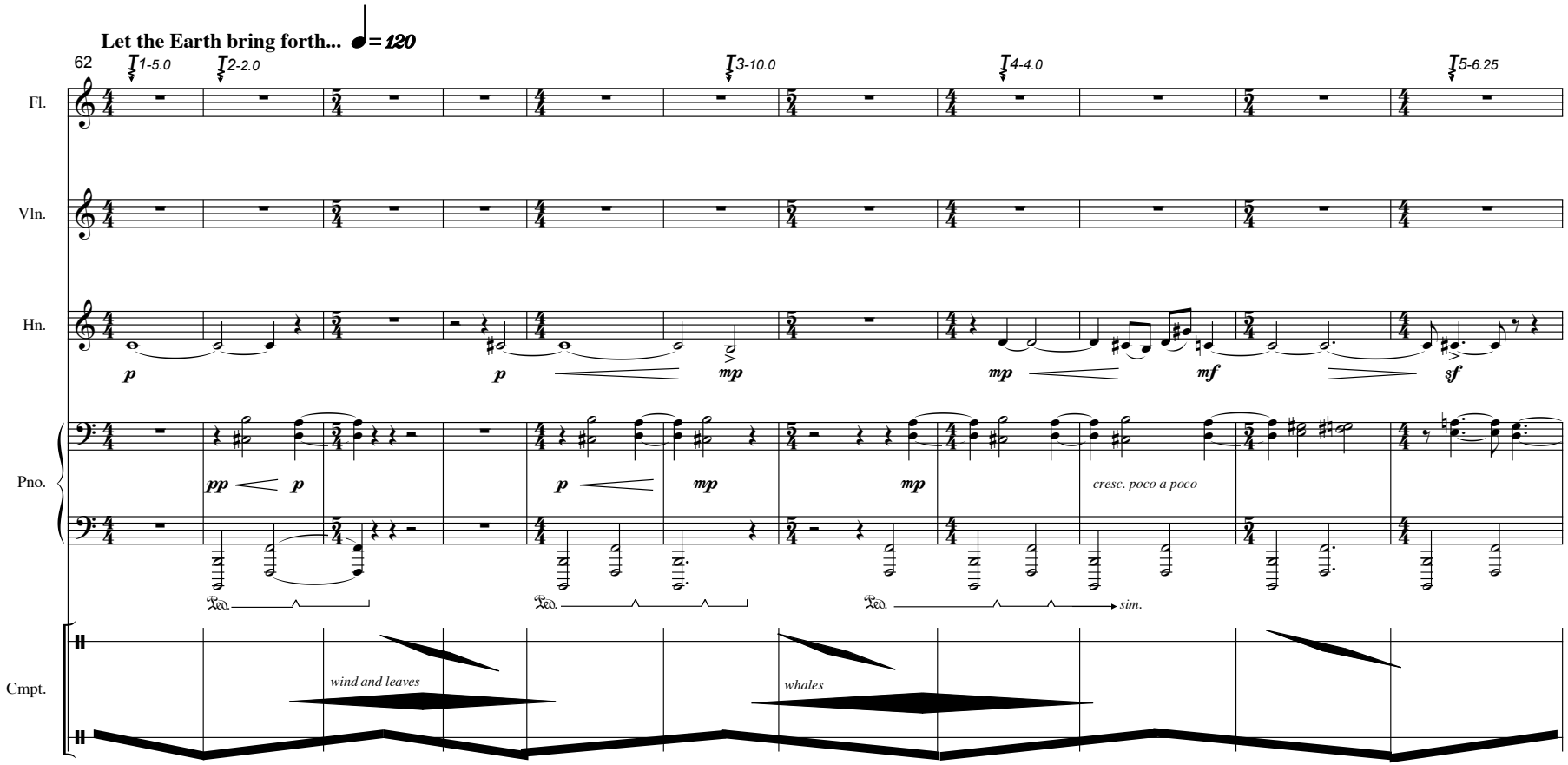

51
Fl.
Vln.
Hn.
Pno.
Cmpt.
mf
ff
fp
mp
f
(Ped.)
10-10.0
11-1.0
12-4.0
13-1.0
(flute delayed harmonizer)
(violin delayed harmonizer)
(add horn +10)
(add piano +10)
(add flute +10)
(wind and water sounds)
rain storm sounds

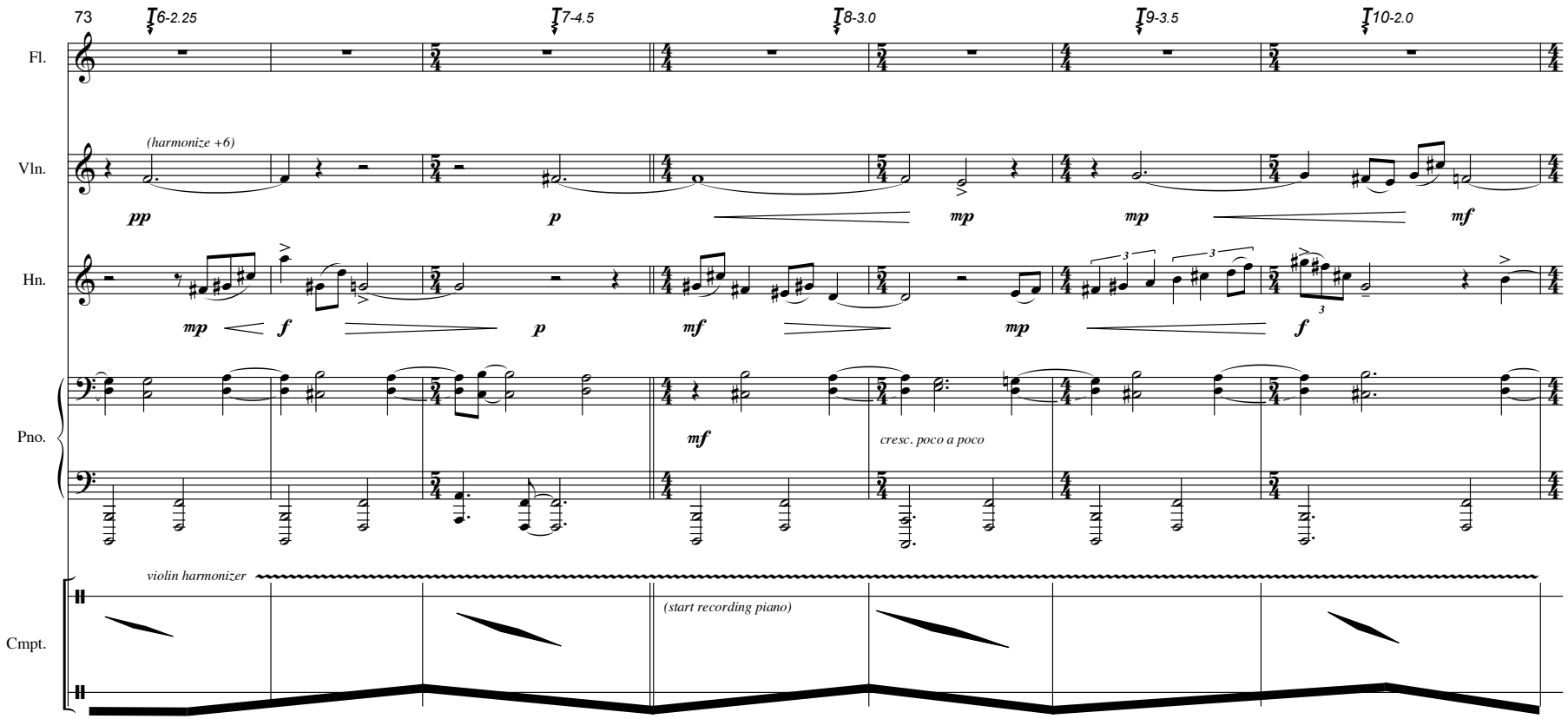

54
Fl.
Vln.
Hn.
Pno.
Cmpt.
(flute, violin, horn, and piano delayed harmonizers)
(wind, rain, and waves)

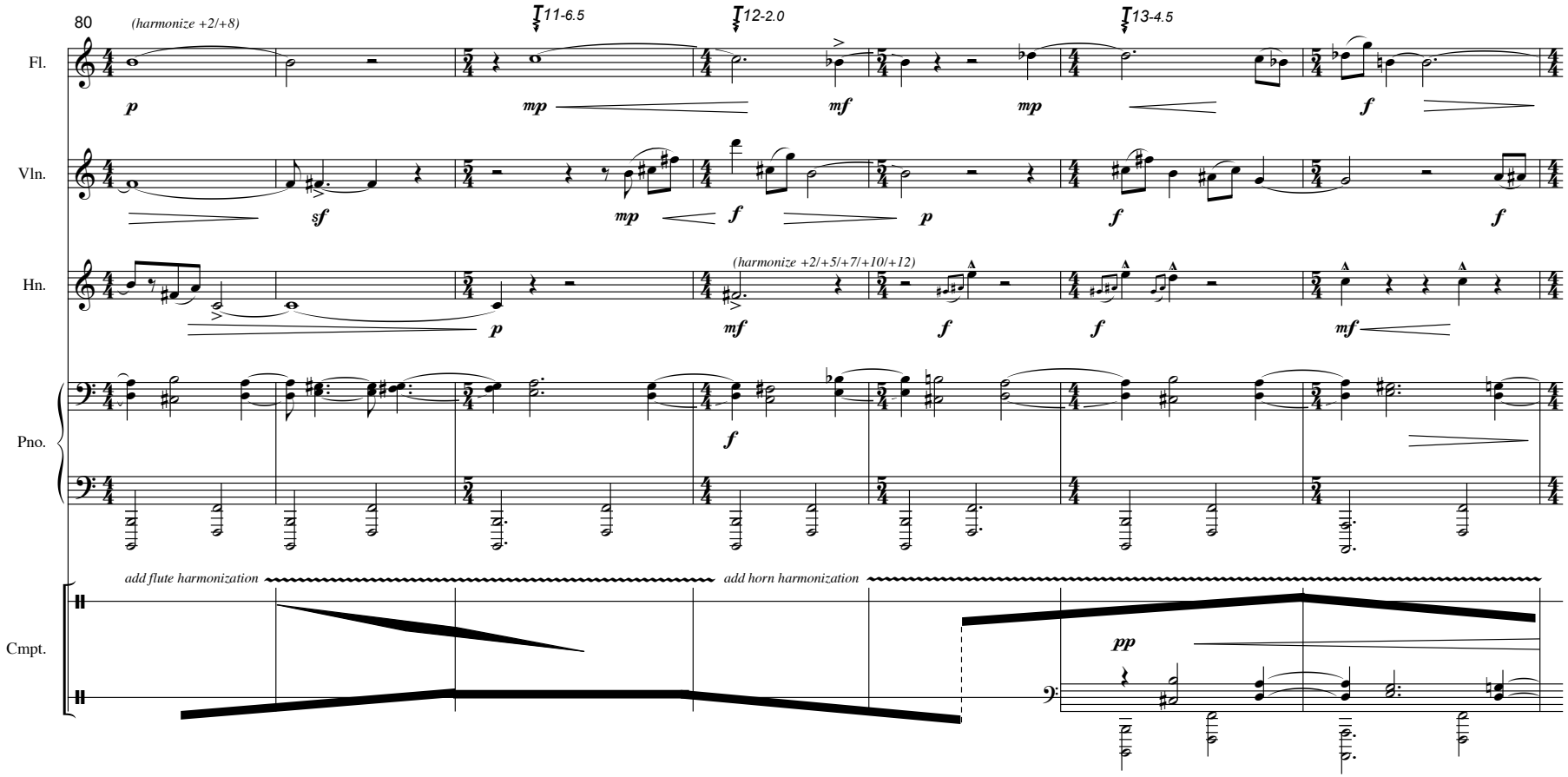

57
ca. 30"
Fl.
Vln.
Hn.
Pno.
Cmpt.
(flute, violin, horn, and piano delayed harmonizers)

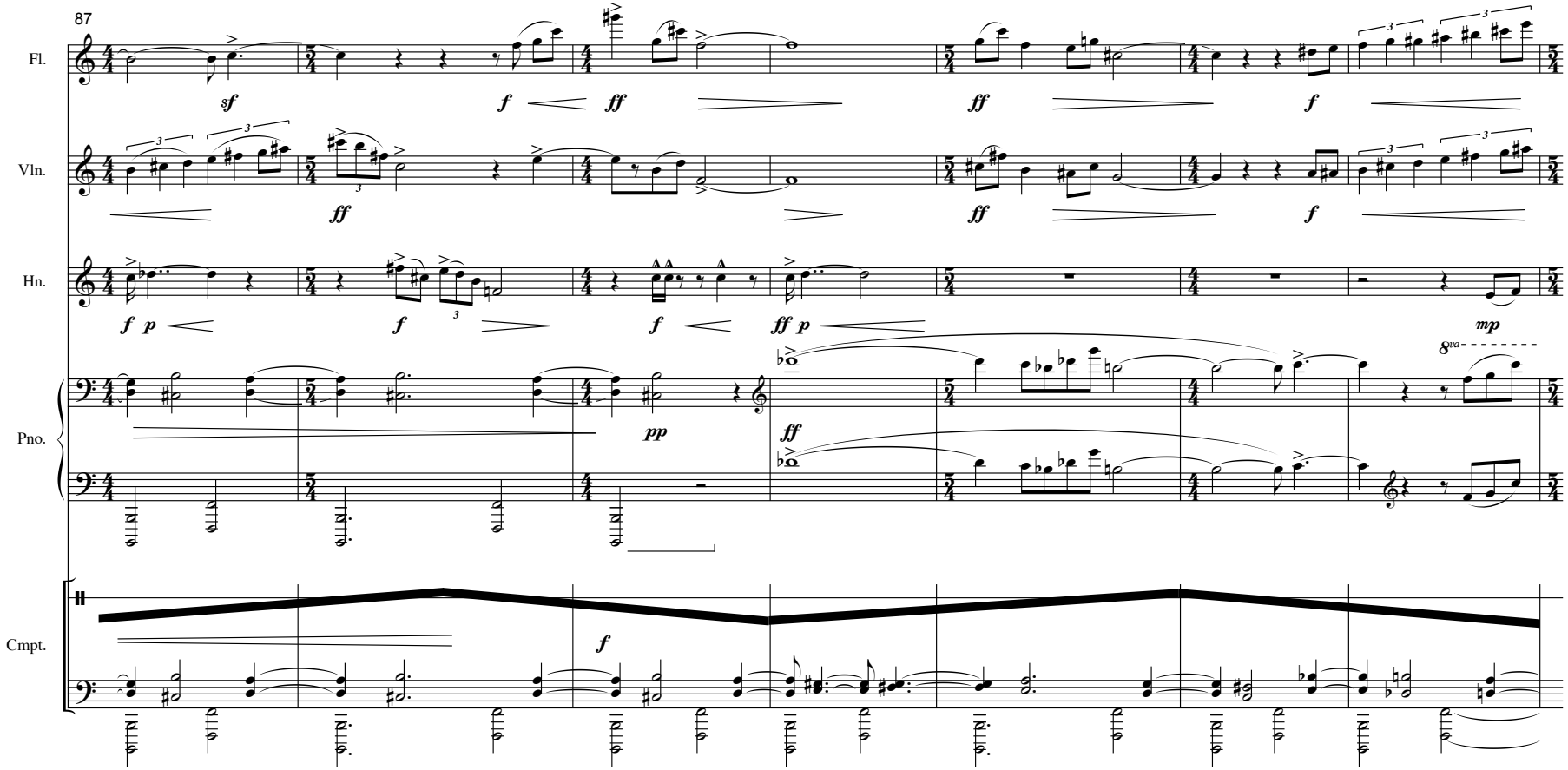

Let the dry land appear...
ca. 9"
58
P1
P2
P3
ca. 12"
Fl.
Vln.
Hn.
Pno.
Cmpt.
ff
f
mf
P4
accel.
rit.
mf
p
f
mp
crash sounds
bouncing sounds
rockslide
swishes

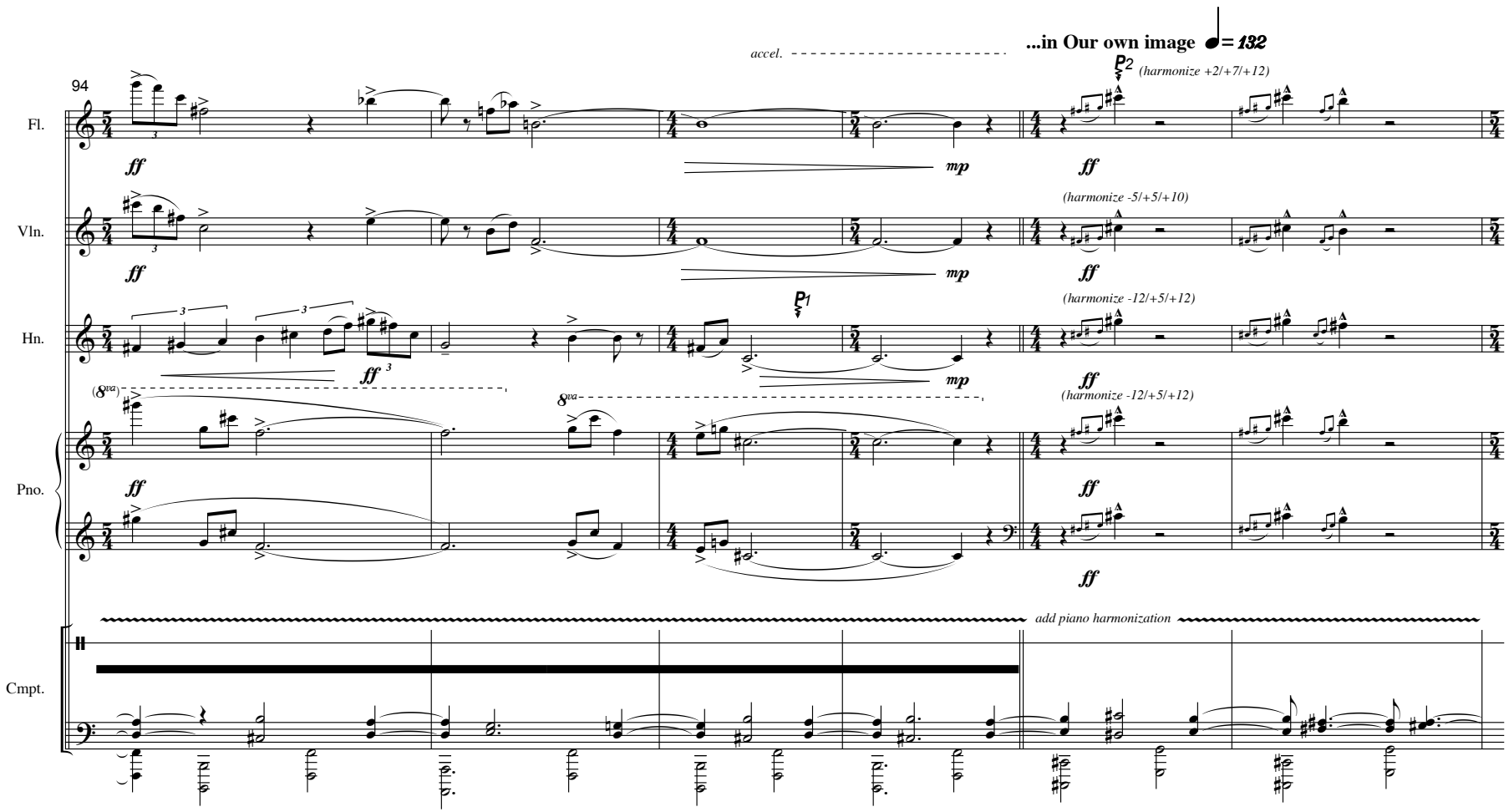

ca. 8"
P5
60
ca. 7"
T7-4.5
Fl.
mp
f
p
P6
f
mp
Vln.
mp
f
mp
f
Hn.
Pno.
p
mp
p
Cmpt.
low continuous sound

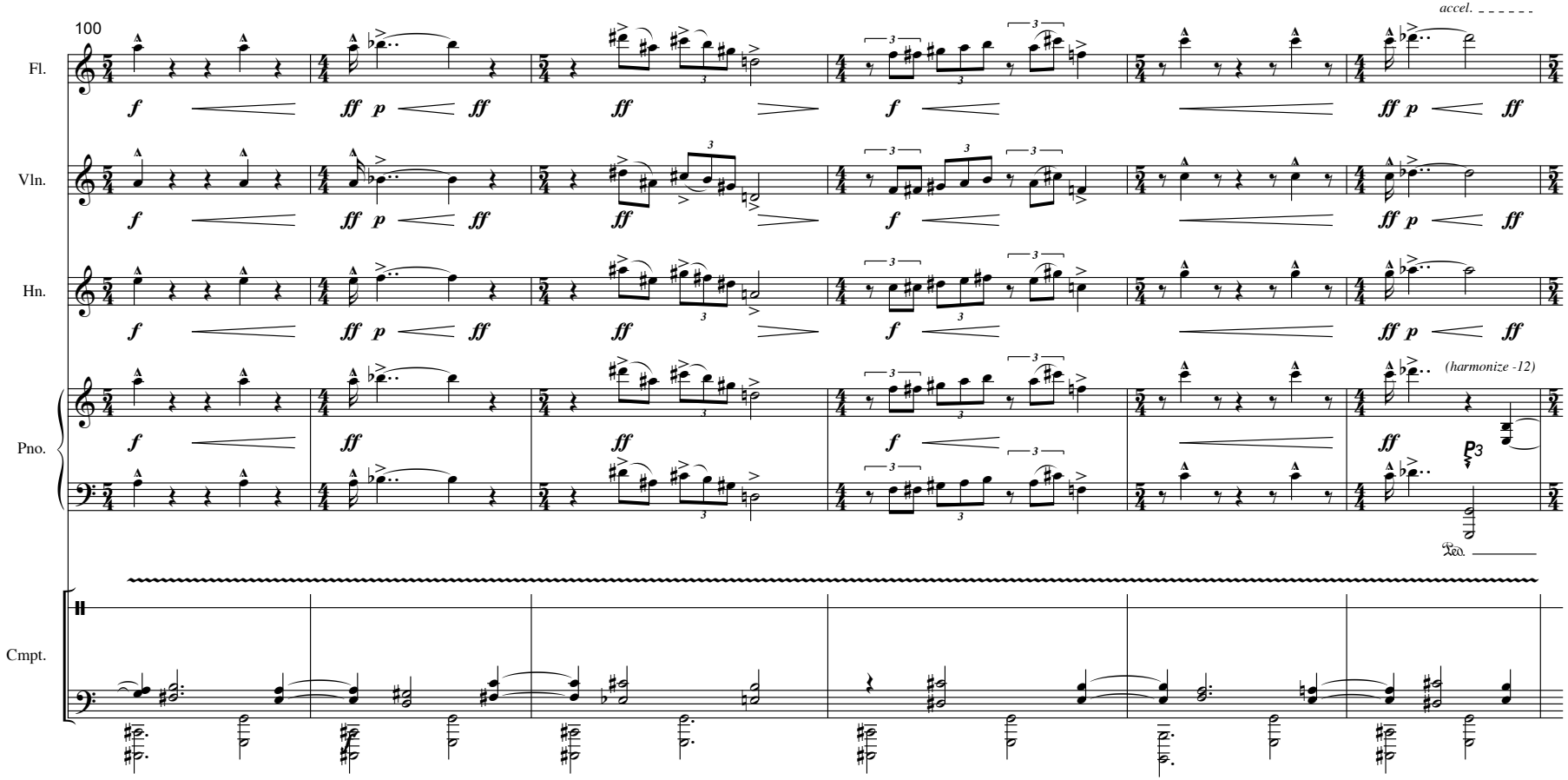

Let the Earth bring forth... ♩= 120
62
1-5.0
2-2.0
3-10.0
4-4.0
5-6.25
Fl.
Vln.
Hn.
Pno.
Cmpt.
p
p
mp
mp
mf
sf
pp
p
p
mp
mp
cresc. poco a poco
Ped.
Ped.
Ped.
sim.
wind and leaves
whales

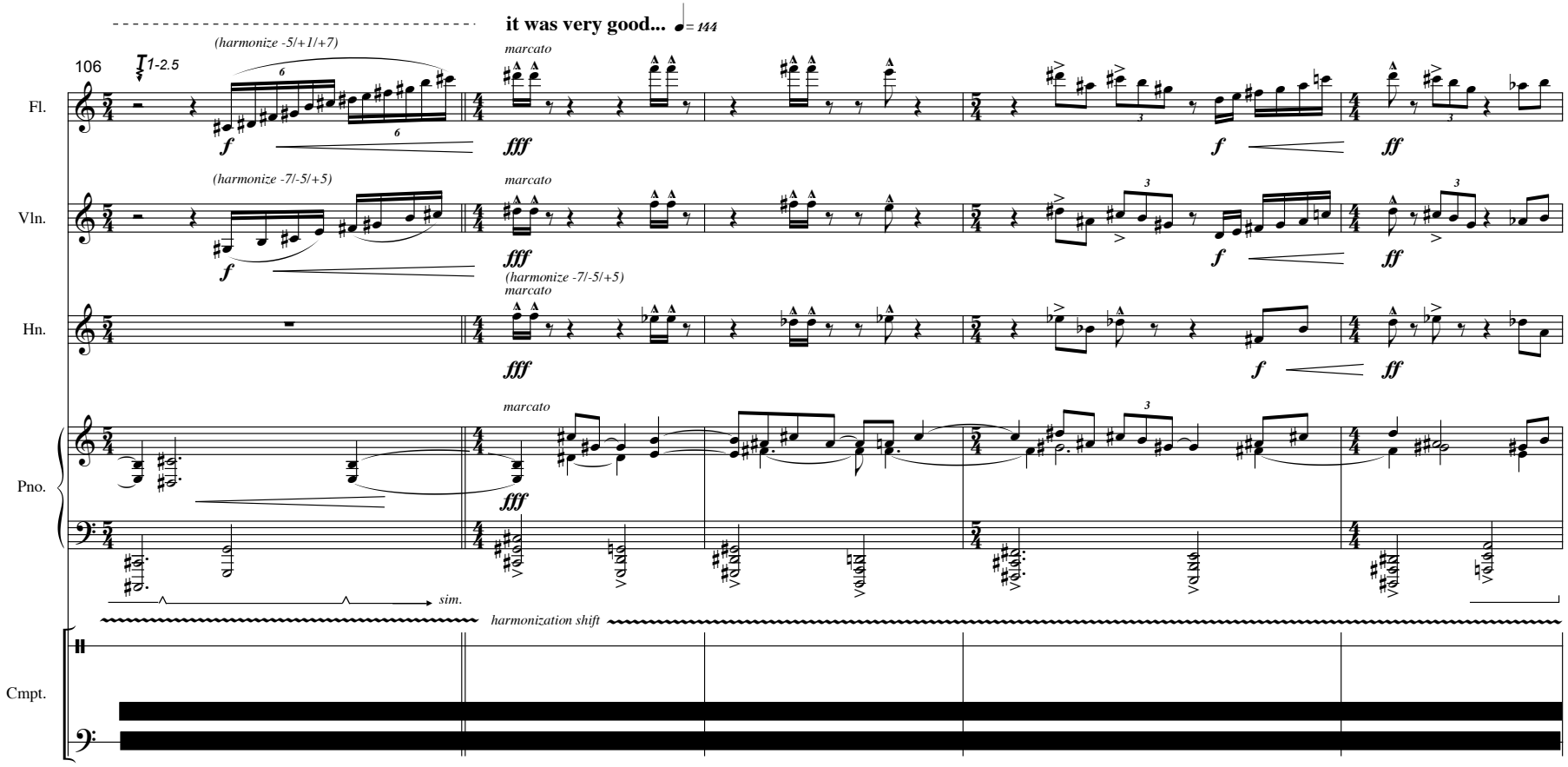

73
T6-2.25
T7-4.5
T8-3.0
T9-3.5
T10-2.0
Fl.
Vln.
(harmonize +6)
pp
p
mp
mp
mf
Hn.
mp
f
p
mf
mp
f
Pno.
mf
cresc. poco a poco
Cmpt.
violin harmonizer
(start recording piano)

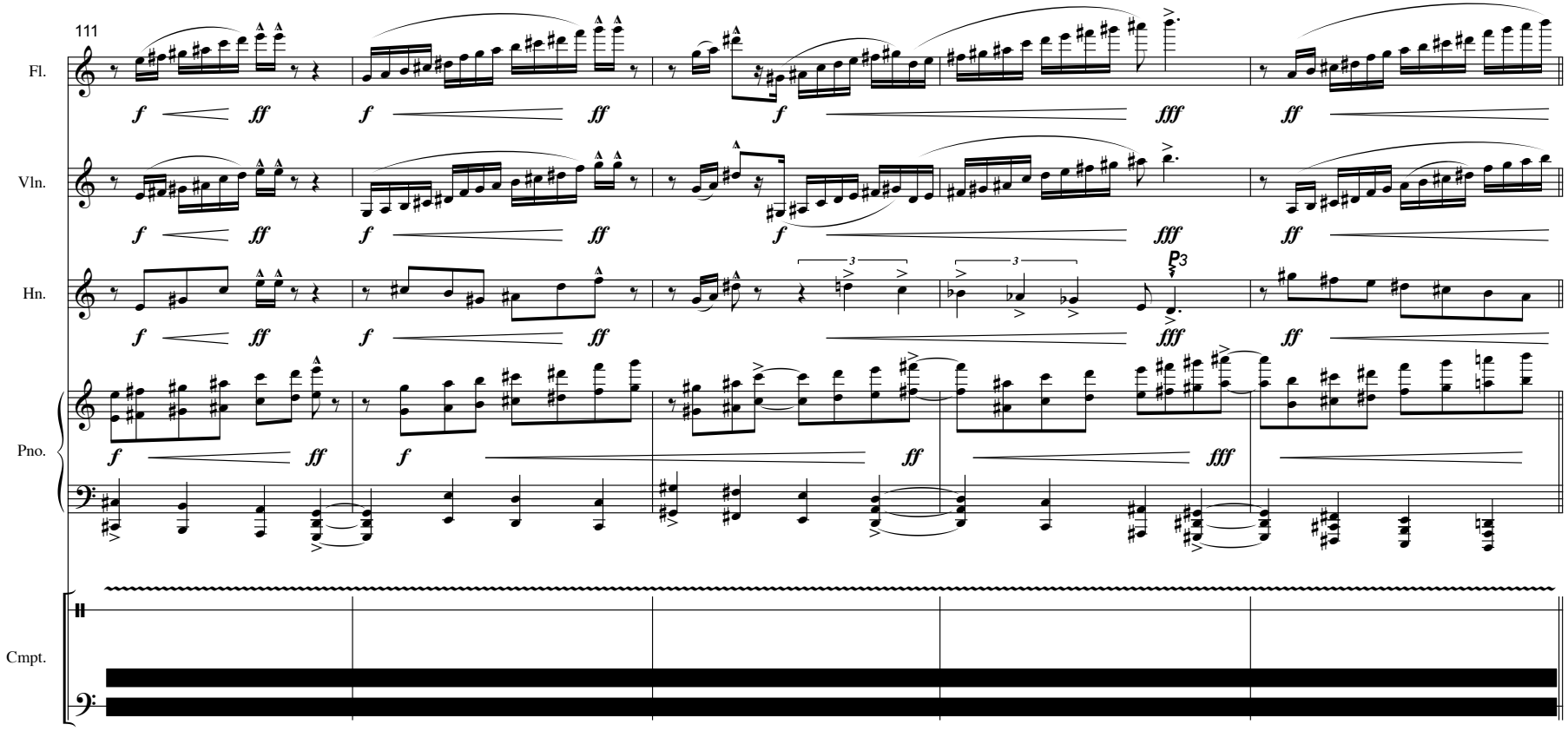

80
(harmonize +2/+8)
11-6.5
12-2.0
13-4.5
Fl.
Vln.
Hn.
(harmonize +2/+5/+7/+10/+12)
Pno.
Cmpt.
add flute harmonization
add horn harmonization

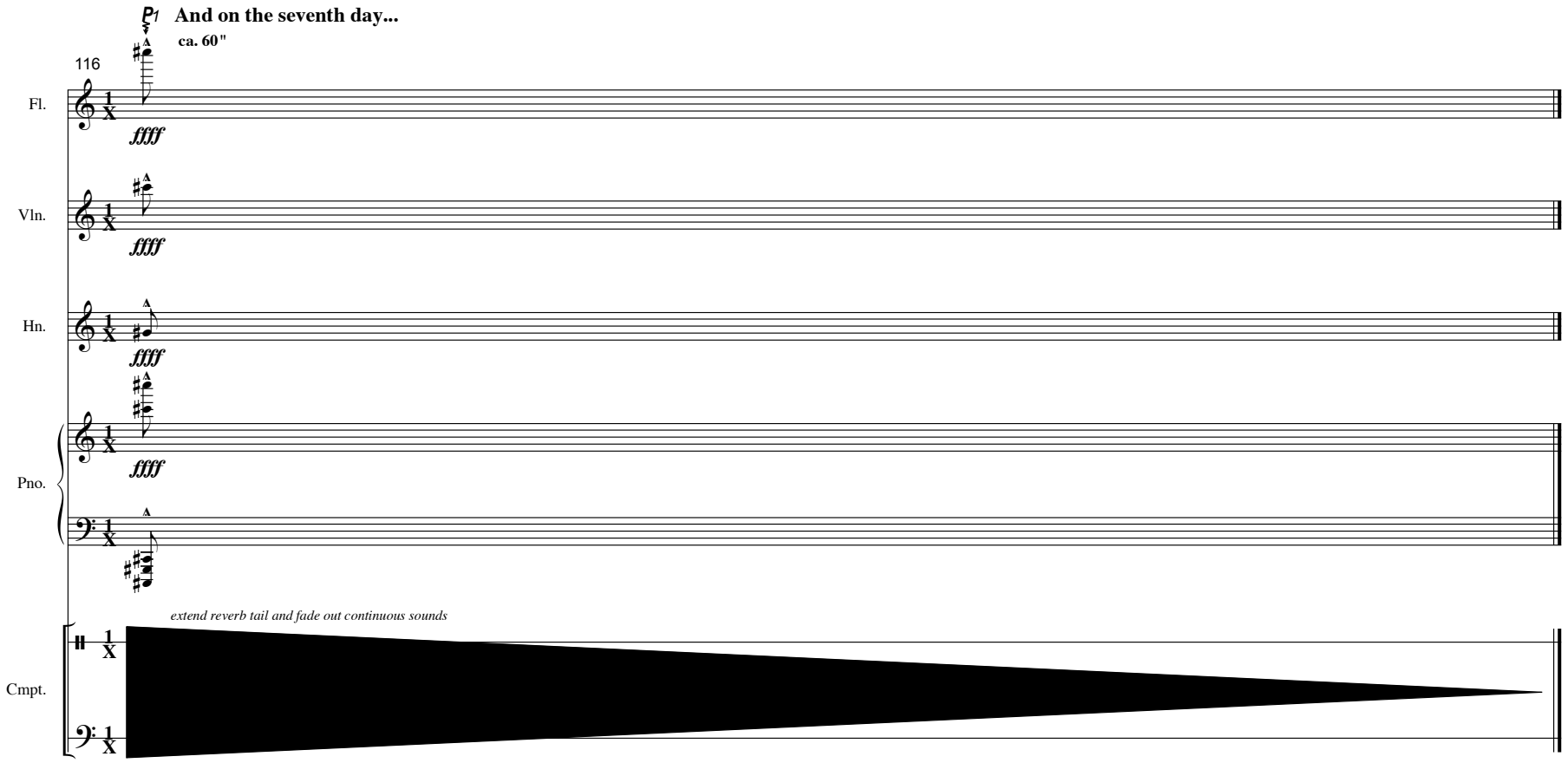

87
Fl.
Vln.
Hn.
Pno.
Cmpt.
sf
f
ff
ff
f
ff
ff
f
f p
f
f
ff p
mp
8va
pp
ff
f


...in Our own image ♩= 132
accel.
94
Fl.
Vln.
Hn.
Pno.
Cmpt.
P2
(harmonize +2/+7/+12)
(harmonize -5/+5/+10)
P1
(harmonize -12/+5/+12)
(harmonize -12/+5/+12)
add piano harmonization


100
accel.
Fl.
Vln.
Hn.
Pno.
Cmpt.
(harmonize -12)
P3
Ped.


it was very good... ♩= 144
106
T1-2.5
(harmonize -5/+1/+7)
(harmonize -7/-5/+5)
(harmonize -7/-5/+5)
marcato
Fl.
Vln.
Hn.
Pno.
Cmpt.
sim.
harmonization shift


111
Fl.
Vln.
Hn.
Pno.
Cmpt.
f
ff
fff
P3


P1
And on the seventh day...
ca. 60"
116
Fl.
Vln.
Hn.
Pno.
Cmpt.
ffff
extend reverb tail and fade out continuous sounds