

oMusink's Tutorial

Introduction

oMusink is a Java tool (requires version Java 1.6 or higher) that links gestures written on paper with online musical objects. It has been based on dot paper technology and digital pens and can work with OpenMusic [1], a visual programming environment for music composition and analysis.

oMusink can be viewed as a customization tool that supports the use, definition and recognition of handwritten gestures. Although paper gestures can take the form of notation elements, oMusink cannot recognize standard musical notation. The primary role of strokes on paper is to describe musical objects and parameters, whose actual digital representation (e.g., a sound, a function) resides on the computer.

oMusink has been designed to support paper-based interactions. Yet, it allows users to input their gestures with a mouse or a tablet. The name of the tool refers to Musink [4], an interaction language for paper musical scores. oMusink is a lightweight version of Musink's gesture browser, supporting a subset of Musink's syntax.

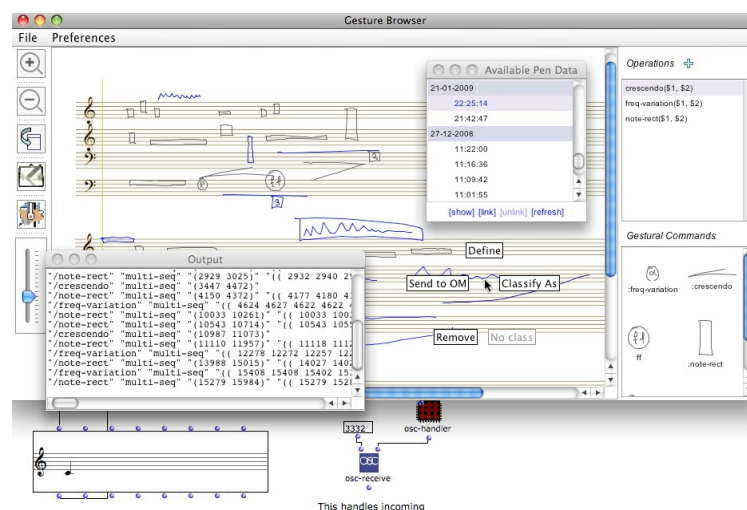


Figure 1. Overview of Musink's user interface

Prerequisites

To install *oMusink*, follow the installation instructions that come with its distribution. In the rest of this document, we assume that the software has been successfully installed. Paper support is not compulsory as users can use the mouse or a tablet to input data.

If you want to use *oMusink* with OpenMusic, you should have it installed in your computer. You can download the last version of OpenMusic from IRCAM's website at <http://recherche.ircam.fr/equipes/repmus/OpenMusic/>

Here, we assume that readers are familiar with OpenMusic.

Running *oMusink* and loading a score

Depending on which platform you use and which installation version you have chosen, the file you need to execute is either *oMusink.app* (Mac OS X) or *oMusink.jar* (cross-platform). Figure 1 shows the interface that you see when you run the application. Note that the first time that you open it, the small window with the pen data may be empty. To open a score, you have to choose Open File from the File menu. Currently, the distribution comes with as single score format that corresponds to the CHORD-SEQ (also MULTI-SEQ) OpenMusic's object (GGFF view) (Figure 3).

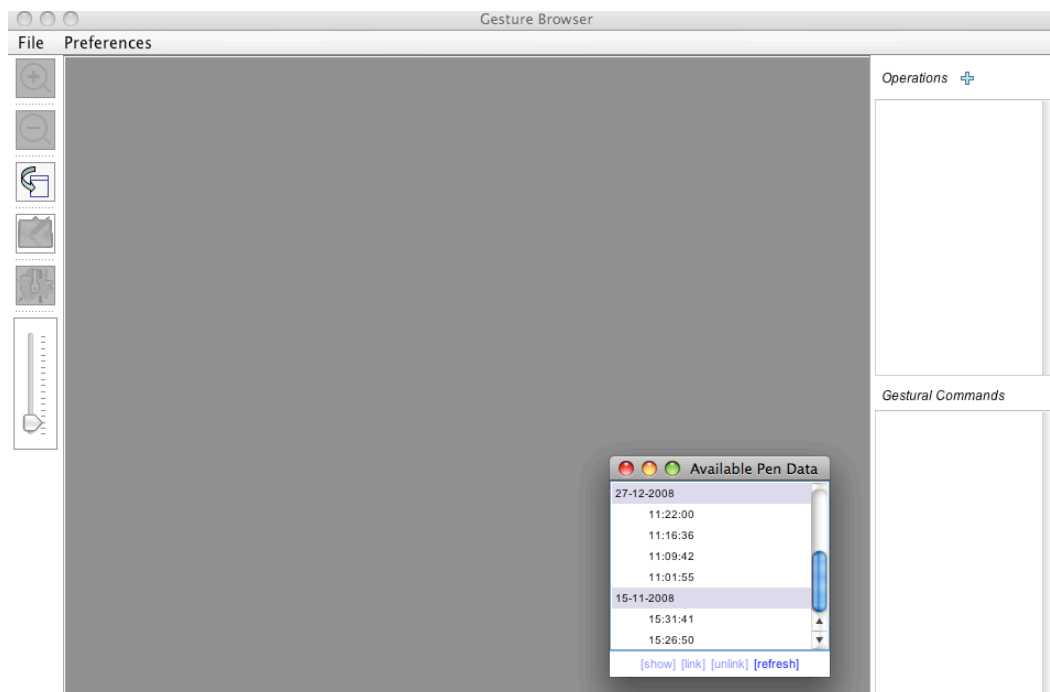


Figure 2. The gesture browser of *oMusink*. A score has not been loaded yet and no gesture definitions exist.

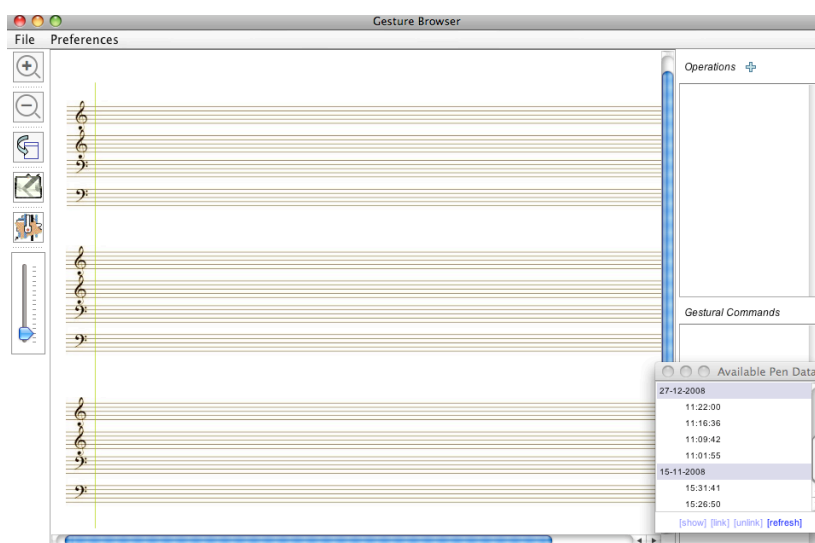


Figure 3. A score that corresponds to GGFF CHORD-SEQ OpenMusic objects.

A PDF of the score (*GGFF.pdf*) can be found under the *scores* directory. This is the file that you should print if you plan to write on the score with a digital pen. Notice that the PDF file is accompanied by text file (*GGFF.txt*). This file describes the coordinates of the score's components on the screen. It is used by oMusink to make the association between the position of gestures written on paper and the score's elements. If you create your own score, you also need to provide a descriptor file (txt with name of PDF file) that describes the structure of the score's elements. Please, contact us to learn more if you plan to create your own score.

Note that there is an option Score templates in the Preferences menu. Score templates are described within *doc-formats.txt* in the root directory. An addition to the score's layout (portrait versus landscape), the file specifies several parameters (offsets, scale, etc.) that control how the score is positioned on printed score and the dot pattern with respect with the online score version. Different printers may print the score on different scales and offsets, so users may have to play with these values and make sure that gestures written on paper will be correctly positioned on the online version of the score.

Staffs in the musical score represent timelines. The position of the gestures on the score is not relative, as in standard musical notation, but absolute, as in OpenMusic's MULTI-SEQ objects. The actual duration of a staff is by default 10 secs, but users can change its value by selecting Preferences-> Set Staff Duration. A group of four staves (GGFF) defines a single timeline that starts from the vertical line and continues to the next group of staves.

Tools

A vertical toolbar at the left side of the interface contains a set of useful tools, explained in Figure 4.

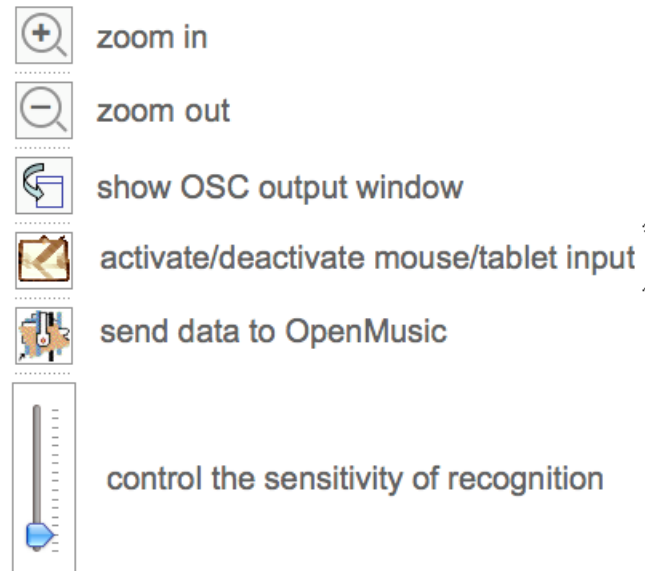


Figure 4. Explanation of the tools in oMusink's toolbar

Entering pen data with a mouse or graphics tablet

To enter data with a mouse or a graphics tablet, the user has to click on the tablet icon on the toolbar and then drag the mouse (or use the pen) on top of the score. When the user clicks on the icon again, the system processes the drawn gestures. In addition, a data entry (identified by date and time) is added to the small pen data window.

Loading previous data

Users can load previously drawn data from the small data window by selecting one or more entries (Click, Shift-Click, Command-Click) from the list and then press on the [show] button at the bottom of the list. The [refresh] button can be used to copy data from an external directory (specified in *config.txt*). The [link] and [unlink] buttons allow users to associate/disassociate an entry of data with a particular score format. If a data entry has been previously associated with a score, a star appears at the right of the time flag that identifies the entry.

Loading pen data from a digital pen

Read the *README-Paper.txt* file that comes with oMusink's distribution to learn how you could connect a digital pen with the tool. Assuming that a digital pen has been successfully connected, pen data will be uploaded to the computer as soon as the user places the pen in the cradle. The configuration file *config.txt* should include the path of the directory where data are uploaded, so that these data can be loaded to the data window (see previous section).

Syntax of supported gestures

The syntax of oMusink's gestures is simple and supports three types of gestures.

1. **Simple one-stroke gestures with an arbitrary shape.** Semantics are not predefined for any gesture so users are free to give their own semantics. You have to expect that if defined gestures have distinguishable visual characteristics (curly shape, vertical versus horizontal, etc.), their recognition will be more accurate.

Each unistroke gesture has three main *properties* that can be communicated to OpenMusic: (1) an identifier (optionally), (2) a time range (x-range), and (3) a list of coordinates (x, y) that represent the points of the gesture's stroke within the spectrum *Time x Pitch*. *Time* is measured in milliseconds and *Pitch* is measured in *midics* (as in OpenMusic). 100 midics corresponds to one semitone. The midic value of G defined by the lower G clef is 6700. The pitch of the lowest line of the F staff in midics is 1900.

The first, central and end points of a gesture are complimentary properties of a unistroke gesture.

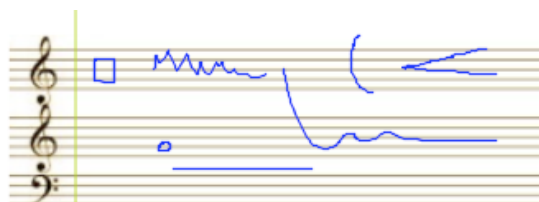


Figure 5. Examples of unistroke gestures

2. **Framed gestures.** They are distinctive from unistroke gestures as they are enclosed within a circle or rectangle and can contain multiple strokes. Their recognition is performed independently from the recognition of unistroke gestures. They can act as regular independent gestures as well as parameters over unistroke gestures (see below). Framed gestures can have the same properties as unistroke gestures (identifiers, list of coordinates of enclosed strokes). Note that the time length (x-range) of a framed gesture

is determined by the range of the frame rather than the time range of the enclosed gestures.

Note: The frame of a framed gesture should be drawn with a single stroke.

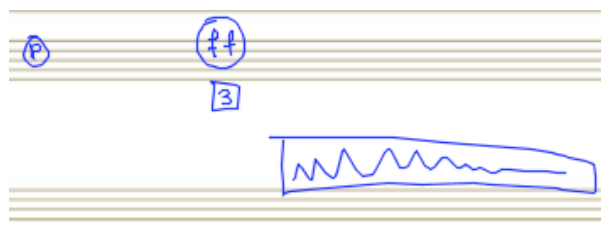


Figure 6. Examples of framed gestures

3. **Complex gestures.** A complex gesture is simply a group of a unistroke gesture and one or more framed gestures. The framed gestures should touch or be in proximity with the unistroke stroke to be considered as part of the group. Examples are shown in Figure 7. In a complex gesture, one of the composing gestures will act as the identifying gesture and the rest will act as parameters. For example, a crescendo from piano (p) to fortissimo (ff), as the one shown in the figure, is identified by the well-known crescendo symbol and is parameterized by two framed gestures. A framed gesture, however, can also act as an identifier.

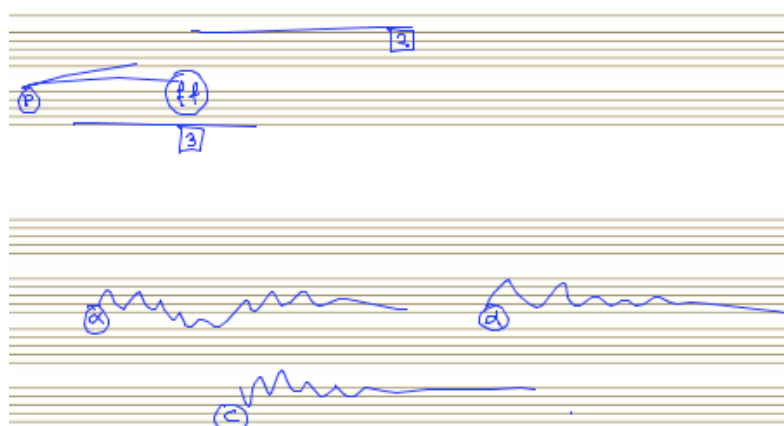


Figure 7. Examples of complex gestures

Specifying operations

oMusink operations serve as interfaces between paper gestures and external applications, OpenMusic in particular. An operation is a function specification (identifier of the function and one or more arguments) that can be later linked to simple or complex gestures. Various types of gesture properties can act as arguments for a given operation:

- Lists of points (x_1, \dots, x_n) (y_1, \dots, y_n) that represent the *(time, midic)* coordinates of the individual points of a unistroke or framed gesture
- The time range (x_1, \dots, x_n) of a unistroke or framed gesture
- The first point *(time, midic)* of a unistroke or framed gesture
- The central point *(time, midic)* of the rectangular gesture's boundaries
- List of strings representing parameters associated with a gesture

Figure 8 demonstrates the definition of a new operation by using oMusink's user interface. In this example, the user defines a new type of tremolo that takes as arguments the points of the stroke that describe the form of the tremolo and its time range (x-range). After the operation has been defined, it appears in the list of operations. Later, the user can modify it by double-clicking on its identifying label.

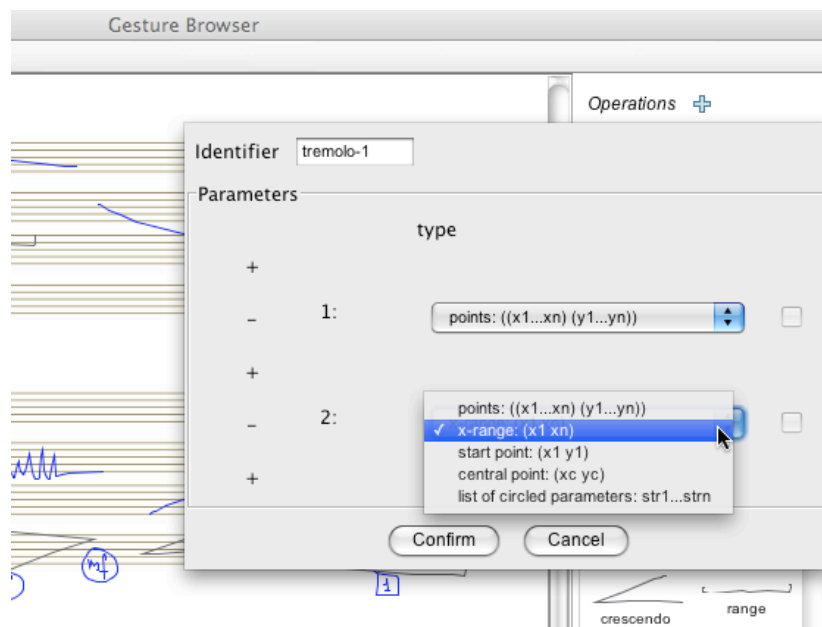


Figure 8. Definition of a new operation

Defining gesture classes

Gestures can be classified into gesture classes that have been previously defined by the actual user. A gesture class can represent a parameter or the identifier of a simple or complex gesture. In the latter case, it can be associated with an operation.

Figure 9 illustrates how the user can activate the definition of a gesture through a contextual menu. As shown in Figure 10, the definition of a gesture class includes a name, a textual description, and optionally an operation. A gesture that has not been linked with an operation could simply act as a parameter. In this

case, its value is represented by the name given in its definition. For example, a circled *f* could represent a parameter with value “*forte*”.

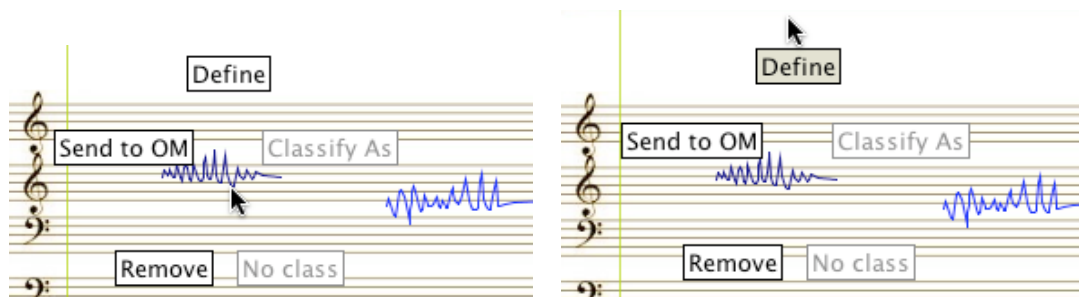


Figure 9. Activating a contextual menu to define a gesture. The menu is activated by right clicking on the gesture and dragging to the direction of the menu option. The option is activated when the mouse button is released.

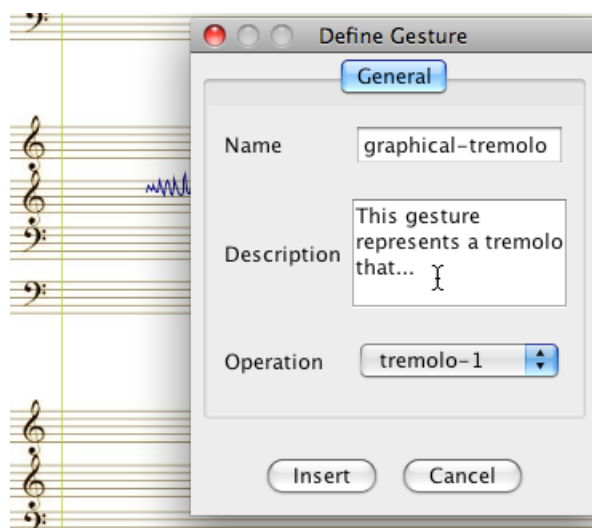


Figure 10. Defining a new gesture class over a unistroke gesture and associating it with an operation.

When the user inserts the new definition, a thumbnail of the gesture is added into the list of defined gestures. Also, the gesture is added into the vocabulary of the gesture recognizer. In this way, other similar gestures on the score will be automatically classified to this class. We use the Rubine algorithm [2] as implemented by the iGesture project [3] to recognize gestures. As recognition is based on a limited collection of available samples, you should not expect the recognition to be perfect. The third tremolo in Figure 11, for instance, has not been recognized. Yet, the user can activate the contextual menu (right click on the gesture) to enforce its classification (see Figure 12). Similarly, the user can use the menu to remove a gesture from a certain class (“No class” menu option).

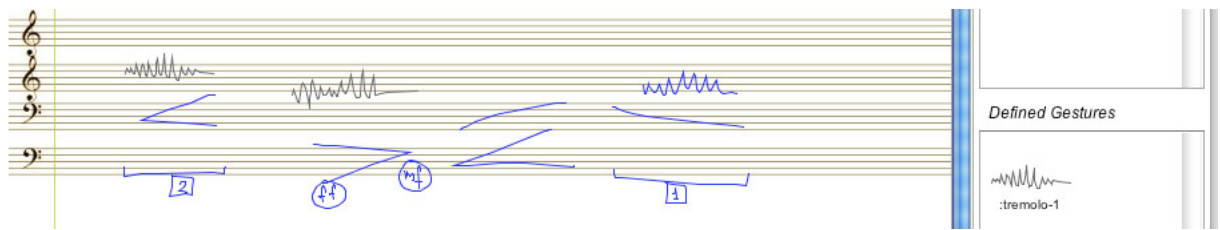


Figure 11. A new tremolo gesture is added into the vocabulary of defined gestures

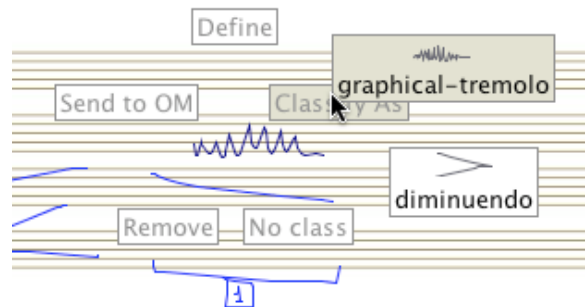


Figure 12. The user activates the contextual menu to enforce the classification of a gesture. The gesture is added to the available samples for the corresponding class.

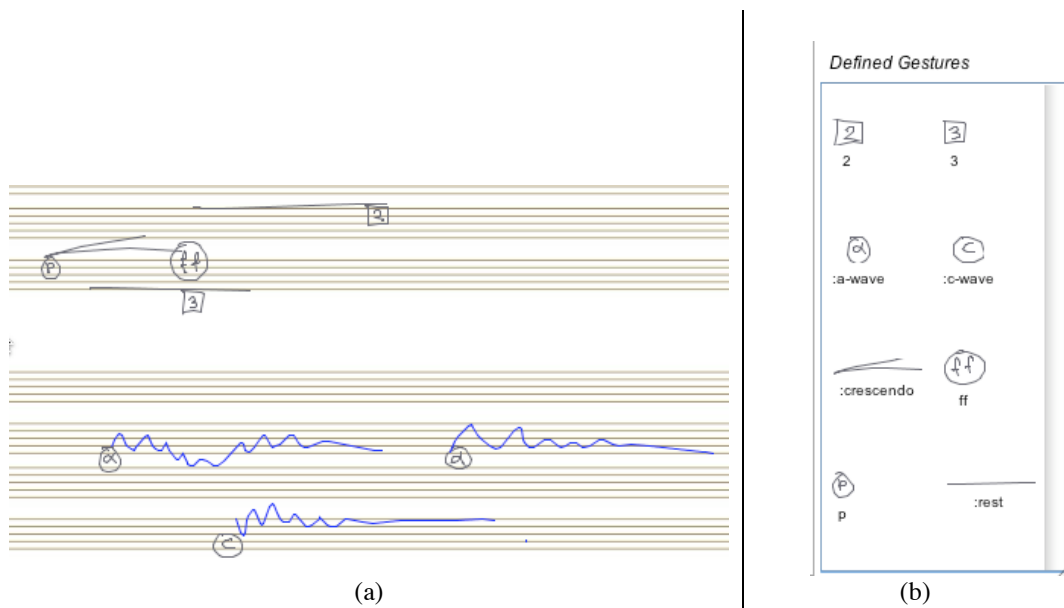


Figure 13. (a) The three top gestures are identified by the unistroke gesture of the group. The framed gestures (*p*, *ff*, 2 and 3) act as parameters: they have been defined, but no operation has been linked to them. The three bottom gestures are identified by circled gestures (*a* and *c*). In this case, *a* and *c* have been linked to two different operations. The unistroke gestures simply act as graphical parameters. (b) The gesture vocabulary that corresponds to the gestures shown at the left.

To define a complex gesture, the user should first choose an identifier. The identifier can be either a unistroke or a framed gesture within the group. Figure 13 explains how a framed or unistroke gesture could be used as identifier of a complex gesture.

Managing gestures

oMusink provides some basic functionality for handling gestures and gesture definitions:

- **Remove a gesture.** Activate contextual menu and select “Remove”
- **Remove a gesture class.** Select a gesture class in the list of defined gestures and press the “delete” key.
- **Modify a gesture class.** Double-click on the thumbnail of the gesture class.
- **Filter gestures on the score.** Select one or more classes of gestures in the list, and press “f”.
- **Export (for future use) definitions of operations and gestures.** Select “Export...” from the File menu and choose a definition file.
- **Import previously exported definitions of operations and gestures.** Select “Import...” from the File menu and choose a definition file.

Sending gesture information to OpenMusic

oMusink can communicate recognized gestures to OpenMusic (or other interested applications) through the Open Sound Control protocol (OSC) [5], a communication protocol based on TCP/IP that is widely used in music technology to connect music applications and devices. For each recognized gesture, an OSC message is sent that communicates the information specified by the associated operation: (1) the name of the operation; and (2) gesture properties (e.g., x-y points of gesture, x-range) that match the operation’s arguments. This information can be sent either individually for each gesture through the contextual menu, or for all the recognized gestures when pressing the button with the OpenMusic logo on the toolbar (see Figure 4). The syntax of the OSC messages sent by oMusink is as follows:

```
/operation-name "multi-seq" arg1 ... argn
```

where `operation-name` is the name of the operation that a gesture is associated with, and `arg1 ... argn` are gesture properties that match the operation’s arguments. Table 1 presents examples of gestures and associated operations. Figure

14 shows the resulting OSC messages as shown on the OSC output window. The OSC output window lets users see what messages are sent through OSC.

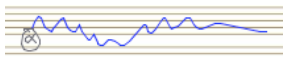



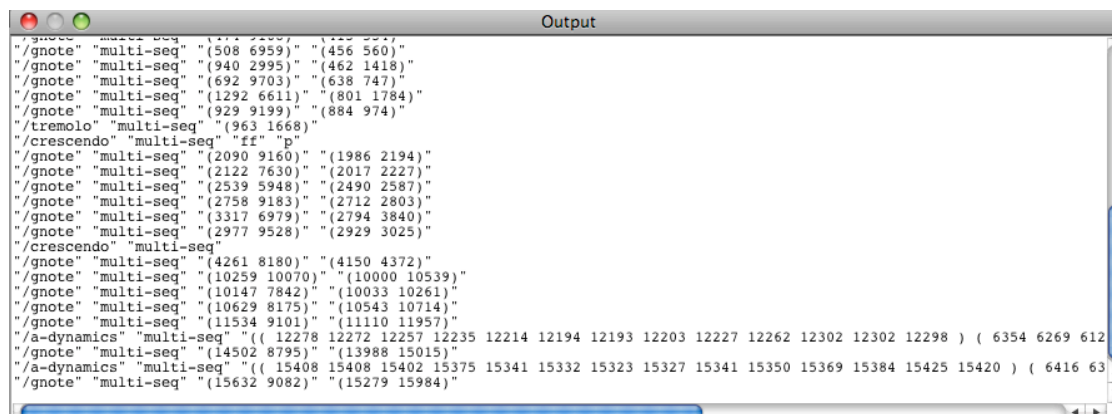
Gesture	Operation
	a-dynamics: <i>points</i>
	gnote: <i>x-range, central-point</i>
	crescendo: <i>list-of-circled-parameters</i>
	tremolo: <i>x-range</i>

Table 1. Examples of gestures and associated operations



```

Output
"/gnote" "multi-seq" "(274 2268)" "(125 554)"
"/gnote" "multi-seq" "(508 6959)" "(456 560)"
"/gnote" "multi-seq" "(940 2995)" "(462 1418)"
"/gnote" "multi-seq" "(692 9703)" "(638 747)"
"/gnote" "multi-seq" "(1292 6611)" "(801 1784)"
"/gnote" "multi-seq" "(929 9199)" "(884 974)"
"/tremolo" "multi-seq" "(963 1668)"
"/crescendo" "multi-seq" "ff" "p"
"/gnote" "multi-seq" "(2090 9160)" "(1986 2194)"
"/gnote" "multi-seq" "(2122 7630)" "(2017 2227)"
"/gnote" "multi-seq" "(2539 5948)" "(2490 2587)"
"/gnote" "multi-seq" "(2758 9183)" "(2712 2803)"
"/gnote" "multi-seq" "(3317 6979)" "(2794 3840)"
"/gnote" "multi-seq" "(2977 9528)" "(2929 3025)"
"/crescendo" "multi-seq"
"/gnote" "multi-seq" "(4261 8180)" "(4150 4372)"
"/gnote" "multi-seq" "(10259 10070)" "(10000 10539)"
"/gnote" "multi-seq" "(10147 7842)" "(10033 10261)"
"/gnote" "multi-seq" "(10629 8175)" "(10543 10714)"
"/gnote" "multi-seq" "(11534 9101)" "(11110 11957)"
"/a-dynamics" "multi-seq" "( ( 12278 12272 12257 12235 12214 12194 12193 12203 12227 12262 12302 12302 12298 ) ( 6354 6269 612
"/gnote" "multi-seq" "(14502 8795)" "(13988 15015)"
"/a-dynamics" "multi-seq" "( ( 15408 15408 15402 15375 15341 15332 15323 15327 15341 15350 15369 15384 15425 15420 ) ( 6416 63
"/gnote" "multi-seq" "(15632 9082)" "(15279 15984)"

```

Figure 14. The OSC output window

Working with OpenMusic

In OpenMusic, the user should create an OSC server to wait for messages.

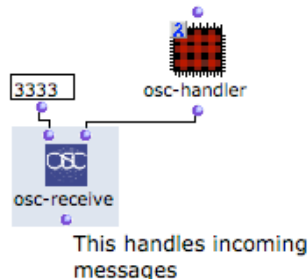


Figure 15. A server handling OSC messages in OpenMusic

Figure 15 illustrates the creation of an OSC server by using OpenMusic's visual language. OpenMusic provides the `osc-receive` function to receive OSC

messages. *oMusink* uses the TCP port 3333 to send OSC messages to the local machine, but you can change its value through the Preferences menu (*Preferences* -> *Configure OM Connection*).

The patch *osc-handler* shown Figure 15 translates incoming messages and creates the objects that correspond to the recognized gestures. It could combine OpenMusic visual elements and LISP code. Figure 16 (a) illustrates a simple implementation of *osc-handler* that simply prints each received message. Note that printed messages can be viewed on the OM Listener. Figure 16 (b) shows an implementation that handles *gnote* and *crescendo* operations. Figure 17 illustrates the LISP code for the *assign-message-handler* function.

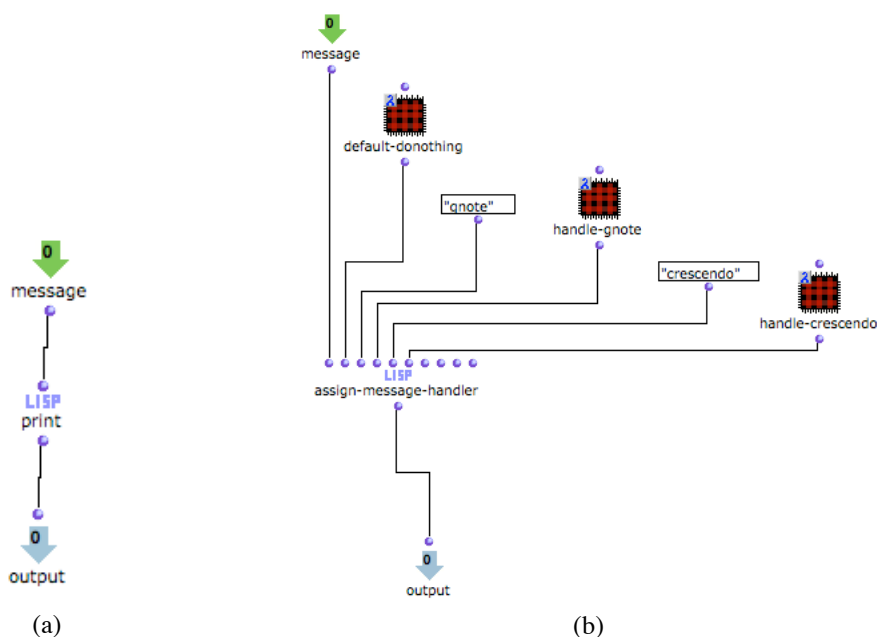


Figure 16. Examples of basic OSC handlers (*osc-handler*): (a) it prints the received message; and (b) it redirects the message to handlers of specific operations (*gnote* and *crescendo*).

```
(defun assign-message-handler (message default_patch str1 patch1 str2
patch2 str3 patch3 str4 patch4)
  (let ((fun (car message)))
    (cond
      ((contains str1 fun)
       (funcall patch1 message))
      ((contains str2 fun)
       (funcall patch2 message))
      ((contains str3 fun)
       (funcall patch3 message))
      ((contains str4 fun)
       (funcall patch4 message))
      (t (funcall default_patch message))))))
```

Figure 17. Example of a LISP implementation of *assign-message-handler*

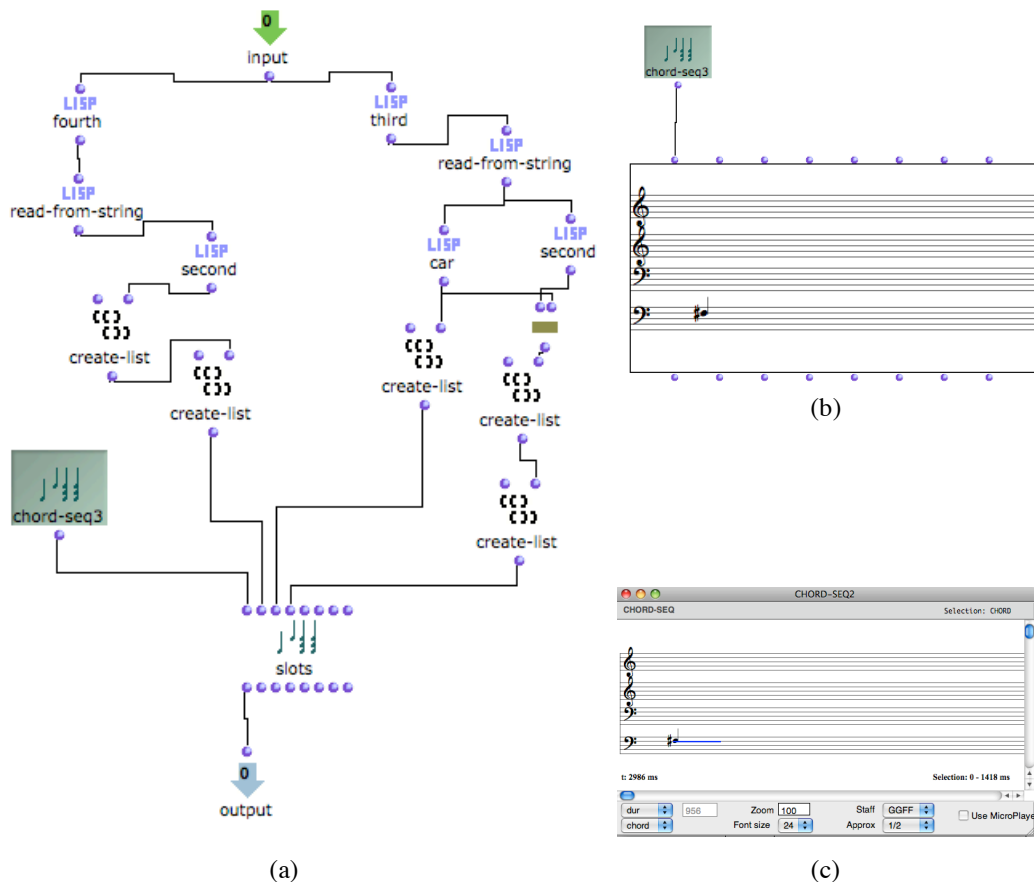


Figure 18. (a) A simple implementation of the *handle-gnote* patch. The *chord-seq3* object is an instance of a CHORD-SEQ object. The slots object has been derived from a CHORD-SEQ object and controls its input (midics, onset times, and durations) based on the values communicated through the OSC message. (b) A copy of the *chord-seq3* object is used as input to a CHORD-SEQ object (c) The CHORD-SEQ object in its GGFF/durations view. Note that the *chord-seq3* object has to be added to the global space of OpenMusic (Library -> globals) so that it can be used out of the scope of the *handle-gnote* patch.

Figure 18 illustrates a simple implementation of the *handle-gnote* patch. It extracts the position in time (onset time), duration, and midic value of the gnote gesture, that has been communicated through the OSC message, and creates its representation in a CHORD-SEQ instance (*chord-seq3*).

Note: An instance of a CHORD-SEQ object can be created by pressing the Command-Shift keys while dragging the *self* output of a CHORD-SEQ object. Also, a *slots* object can be created by pressing the Shift key while dragging a CHORD-SEQ object.

Unfortunately, the above implementation does not allow for showing multiple *gnote* gestures on a CHORD-SEQ object. This requires some additional pro-

gramming. A solution is shown in Figure 19. An example of a CHORD-SEQ object created in this way is shown in Figure 20.

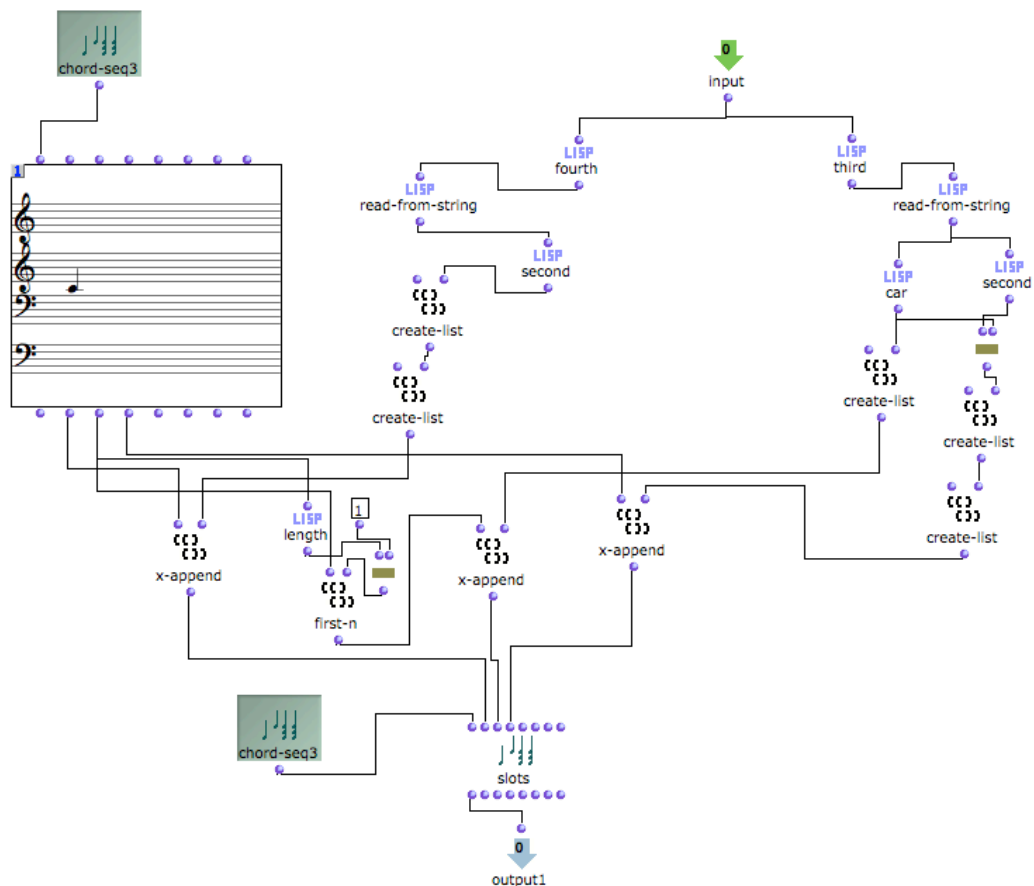


Figure 19. Implementation of *handle-gnote* that allows for expressing multiple *gnote* gestures on a CHORD-SEQ object. The *chord-seq3* object changes incrementally every time the patch is executed (a new note is added each time).

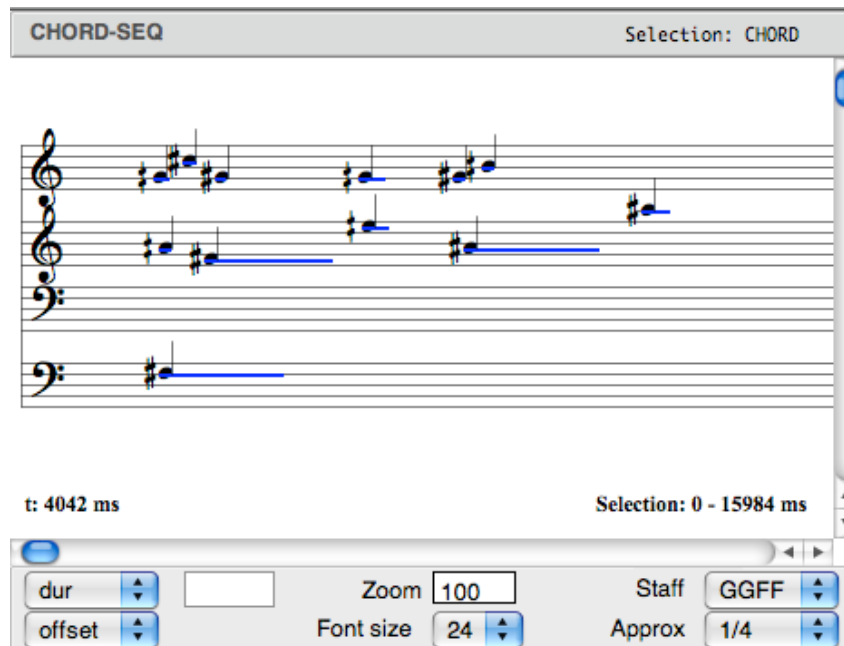


Figure 20. A CHORD-SEQ object created from the patch shown in Figure 19. Each note has been the result of a single *gnote* gesture.

More information

For more information, requests and problems, contact Theophanis Tsandilas.
Email: fanis at lri dot fr

Acknowledgments

Catherine Letondal and Wendy Mackay have participated in the conceptual design of oMusink. Carlos Agon has indicated the OSC protocol for the communication between oMusink and OpenMusic. He also helped us establish the connection between OSC messages and OpenMusic objects. Mikhail Malt has provided valuable feedback and suggestions concerning the use of the CHORD-SEQ object in connection with pen and paper.

References

- Agon, C., G. Assayag, and J. Bresson, *OM Composer's Book*. 2006: Editions Delatour France, Ircam.
- Rubine, D. Specifying gestures by example. In *Proc. SIGGRAPH 1991*, ACM Press (1991), 329-337.
- Signer, B., U. Kurmann, and M.C. Norrie. iGesture: A General Gesture Recognition Framework. In *Proc. ICDAR (2007)*, 954-958.
- Tsandilas, T., C. Letondal, and W.E. Mackay. Musink: composing music through augmented drawing. In *Proc. ACM CHI (2009)*, 10 pages.
- Wright, M. and A. Freed. Open Sound Control: A New Protocol for Communicating with Sound Synthesizers. In *Proc. ICMC (1997)*, 101-104.