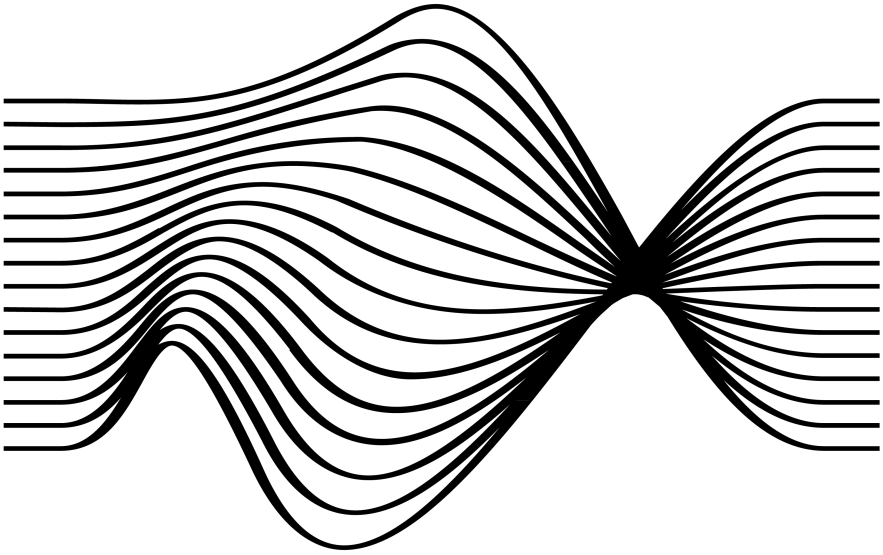

iannix

software documentation

v. 0.9.20

This document is based on:

Scordato, J. (2016). *An introduction to IanniX graphical sequencer*. Master's Thesis in Sound Art, Universitat de Barcelona.

Index

1. What is IanniX? p. 1
    1.1 Main applications p. 2
    1.2 Classification of IanniX scores p. 3
2. User interface p. 5
    2.1 Main window p. 5
        2.1.1 Visualization p. 6
        2.1.2 Creation p. 7
        2.1.3 Position p. 8
        2.1.4 Inspection p. 9
    2.2 Secondary windows p. 10
        2.2.1 Script editor p. 11
3. IanniX objects p. 15
    3.1 Curves p. 16
    3.2 Cursors p. 17
    3.3 Triggers p. 19
4. Control and communication with other devices p. 21
    4.1 Sent messages p. 21
        4.1.1 Cursor-related variables p. 22
        4.1.2 Trigger-related variables p. 24
        4.1.3 *Transport* variables p. 25
    4.2 Received commands p. 25
        4.2.1 Score instances and viewport p. 25
        4.2.2 Object instances and identification p. 26
        4.2.3 Object positioning p. 27
        4.2.4 Object appearance p. 28
        4.2.5 Object behavior p. 29
        4.2.6 Sequencing p. 30

4.3 Network protocols p. 31
4.3.1 OSC p. 31
4.3.2 Raw UDP p. 33
4.3.3 TCP and WebSocket p. 34
4.3.4 HTTP p. 35
4.4 MIDI interface p. 36
4.5 Serial interface p. 37
4.6 Software interfaces p. 38
4.6.1 Recursivity p. 38
4.6.2 Syphon p. 40
References p. 41
Annex 1: JavaScript Library p. 45

1. What is IanniX?

IanniX is a graphical sequencer for digital art. Through various communication protocols (cf. Chap. 4), it synchronizes single events as well as continuous data to external environments (e.g. Pure Data and Processing) and hardware such as MIDI devices and microcontroller boards.

Its graphical interface shows a representation of a multidimensional and multi-format score which is programmable via GUI, JavaScript and third-party applications that use a compatible protocol; in this way, users are not forced to a specific method for approaching to score but can benefit of multiple designing strategies according to their expertise. Such interface is based on three types of abstract objects to be placed in a three-dimensional space: triggers, curves, and cursors (cf. Chap. 3). Triggers and curves represent respectively single events and spatial trajectories. Cursors are time-based elements – playheads – that can move along curves in order to read a specific sequence of space-limited events. An undetermined number of cursors can be added to the score. In this sense, IanniX proposes a three-dimensional and poly-temporal sequencer, unlike its predecessor UPIC (Raczinski, Marino and Serra, 1990; Bourotte, 2012) that was based on bi-dimensional writing and allowed for only a single timeline from left to right, as an emulation of the conventional direction of reading. Also, IanniX runs independently from any audio synthesis engine; thus, it is suitable for varied artistic applications.

IanniX software is free, open-source, and cross-platform, in order to reach almost the whole community of computer users without significant limitations. The software package is available for download at www.iannix.org.

1.1 Main applications

Through the communication with audio environments or MIDI devices, IanniX can be used as a tool for the creation and the performance of musical scores with a graphic notation. Many object attributes as well as various mapping modes (cf. Chap. 3.2) allow the user to match the characteristics and the behavior of cursors, curves, and triggers to sound and music parameters and several MIDI messages (cf. Chap. 4.4). The option of importing external graphics amplifies the representational possibilities of basic objects; furthermore, sketches and notes can be integrated into the score for the definition of the final result. Scores can be also generated partially or entirely by script, thus adding a further level of complexity. Several examples are contained into IanniX software package, including the score of *Recurrence* (2011) by Thierry Coduys and an excerpt from *Metastaseis* (1953-54) by Iannis Xenakis.

The strong relation between sound and visual content that emerges by the use of IanniX has been often a stimulus to reveal the score as an integral part of the work ("Showcase | IanniX", 2017). In this sense, IanniX has been used in audiovisual works for controlling audiovisual parameters and showing their representation to the public, even to facilitate the formal intelligibility (Alessandretti and Sparano, 2014; Scordato, 2015).

Specific usages of IanniX include the control of sound spatialization, both for the definition of virtual sound trajectories ("World Expo 2012 Korea | IanniX", 2012; Ranc, 2013; Manaris and Stoudenmier, 2015) and the routing of audio signals in complex sound projection systems. Moreover, IanniX has been employed in sonification processes (Bellona, 2013).

1.2 Classification of IanniX scores

In relation to their functionality and to the interaction mode with third-party software and hardware, different score types have been recognized (Coduys and Jacquemin, 2014):

- *control score*; it is autonomous, reproducible, and determinist; similarly to the operation of UPIC, once the score has been set, the sequencer produces a data output for controlling further processes (e.g. sound synthesis, sampling, and spatialization);
- *reactive score*; it reacts to external stimuli without generating any output; primary purposes are visualization and graphical representation of functions or data received from programming environments and devices (e.g. the deformation of a curve expressed by a parametric equation or a three-dimensional trajectory detected by a motion capture device);
- *generative score*; it is produced by algorithms written in JavaScript ("JavaScript | MDN", 2016) and can evolve either in a predetermined way or not; therefore, the script generates the score as output (cf. Chap. 2.2.1);
- *interactive score*; being based on human-computer interaction or software interaction, it involves the cooperation between various entities; in this context, IanniX may act as a mapping device but also introduces compositional and temporal dimensions; a bi-directional data flow is involved;
- *recursive score*; IanniX can control itself, that is to say the output related to an object can be received as input command for controlling either the sequencer or another object; in some cases, this may imply feedback or deformation of the score as a function of time (cf. Chap. 4.6.1).

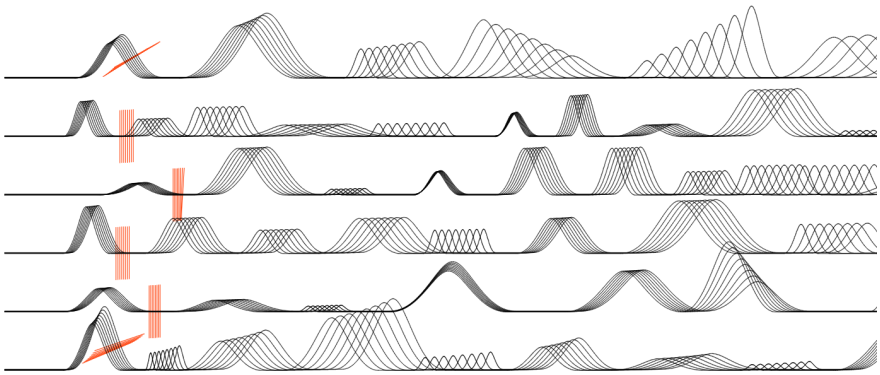

Fig. 1. Excerpt from *Recurrence* (2011) by Thierry Coduys.

2. User interface

IanniX user interface comprehends a main window with operative functions that allow the user to create, edit, and play their own scores. Additionally, a series of secondary windows is supplied for visualization purposes, editing, and support.

2.1 Main window

Based on a user-friendly graphical approach, it represents the basic interface between user and score. Most actions can be carried out through this window, which is divided into four parts:

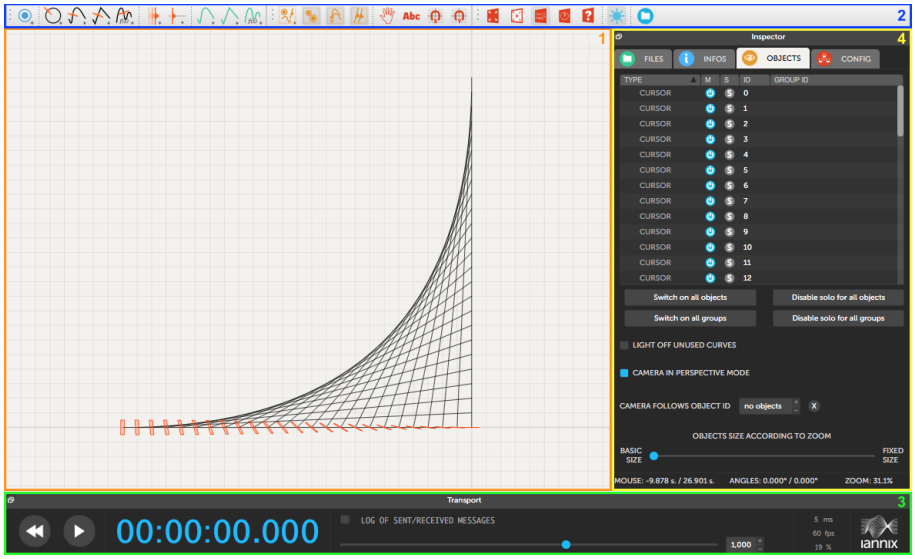

Fig. 2. Main window divided into four sections.

1. *visualization*; the score rendering area consists in a representation of a three-dimensional space for viewing and manipulating IanniX objects directly;
2. *creation*; a toolbar allows for drawing and adding default objects to the score and for setting up several display options;

3. *position*; this panel includes transport controls and performance settings;
4. *inspection*; it shows file browser, object list, attributes, resources, and configuration.

2.1.1 Visualization

Users can navigate throughout the score by means of mouse (or trackpad) and keyboard:

- `click+drag` to scroll in the score;
- `mouse wheel` to zoom in and out;
- `alt+click+drag` to rotate the viewport;
- `alt+mouse wheel` to change the viewport distance;
- `alt+double click` to reset the viewport position.

Score editing functions are available when the viewport is set in its default position (i.e. without rotation):

- `click` to select a IanniX object;
- `shift+click` to select multiple objects;
- `click+drag` to move the objects;
- `ctrl+alt+drag` to enable grid snapping (`cmd key` on Mac).

In order to facilitate objects positioning, grid and axes are displayed as a space-time reference; by default, a grid unit corresponds to one second in time sequencing. Grid resolution and display options can be customized from the *Menu bar* / *Display*. Also, selected objects can be aligned and distributed automatically from the *Menu bar* / *Arrange objects*.

2.1.2 Creation

IanniX objects (cf. Chap. 3) can be drawn and added directly to the score by selecting them from the *Objects creation* toolbar:

Fig. 3. Objects creation toolbar.

1. adds a trigger;
2. adds a circular curve with a cursor;
3. draws a smooth curve with a cursor;
4. draws a straight curve with a cursor;
5. adds a parametric curve with a cursor;
6. adds a cursor on selected curves;
7. adds a timeline;
8. draws a smooth curve;
9. draws a straight curve;
10. adds a parametric curve.

In drawing mode, a `click` on the score sets a point of the curve; `esc` exits this mode. After creating a curve, the user can also modify it: `double click` on the path to add a point on the curve; `ctrl+double click` to remove it (`cmd key` for Mac users); `double click` on a point to enable or disable smoothing.

Cursors and triggers are capable of sending messages through various protocols (cf. Chaps. 4 and 5); `double click` on an object to open the *Message editor*. For testing purposes, with `shift+double click` the user can force a selected object to send its messages.

In addition to default IanniX objects, the user can import background images, text and SVG files (to be converted into IanniX

curves) from the *Menu bar* / *File*.

Further features are available from the *View options* and *Window options* toolbars:

Fig. 4. View and window options.

1. restricts message output capability to selected objects;
2. enables or disables trigger selection;
3. enables or disables curve selection;
4. enables or disables cursor selection;
5. locks objects position to avoid accidental changes;
6. shows or hides object labels;
7. snaps mouse actions to vertical grid;
8. snaps mouse actions to horizontal grid;
9. shows score (*Render*) in fullscreen;
10. enables or disables the *Render* in a separate window;
11. shows or hides the *Script editor* (cf. Chap. 2.2.1);
12. enables or disables the *Timer* in a separate window;
13. shows or hides the *Helper* window;
14. changes IanniX color theme (light or dark);
15. opens *Patches* folder containing interfacing examples.

2.1.3 Position

The *Transport* panel includes the main sequencing controls for moving along the score (i.e. play, stop, and fast rewind) and for changing the global playback speed through a multiplication factor (cf. Fig. 5). It also integrates IanniX performance settings:

- an instantaneous message log that displays the last message

sent or received;

- the scheduler period, which is the interval between two computed events; its default value is 5 milliseconds, but can be adjusted by the user with the awareness that custom values considerably affect IanniX performance, messages accuracy, and CPU usage;
- the rendering frame rate, that is the refresh speed of the displayed score; values between 30 and 50 frames per second are reasonable;
- the CPU usage; processor load can be optimized by disabling message logs and object labels, and by adjusting scheduler period and rendering frame rate.

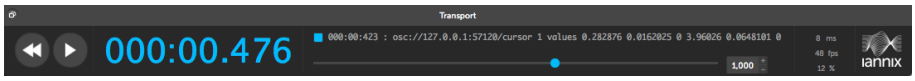

Fig. 5. *Transport* panel.

2.1.4 Inspection

The *Inspector* panel is an essential tool for the management of IanniX scores and objects, and for the configuration of communication interfaces (cf. Fig. 6). It is subdivided into four tabs:

- *FILES*; a file browser for the management of IanniX scores;
- *INFOS*; through this section, users can visualize and edit object attributes (cf. Chap. 3) as well as global colors and textures; features are ordered into five further tabs: *General*, *3D Space*, *Time*, *Messages*, and *Resources*;
- *OBJECTS*; this shows a list of all objects included in the current score, and also permits selection, *mute* and *solo* functions for each object (cf. Fig. 2);
- *CONFIG*; a section which comprehends a full message log as

well as the configuration of supported interfaces: *Network* (cf. Chap. 4.3), *MIDI* (cf. Chap. 4.4), *Arduino* (cf. Chap. 4.5), and *Software* (cf. Chap. 4.6).

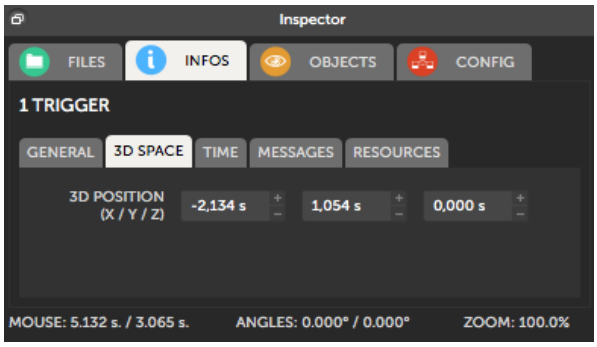

Fig. 6. *Inspector* panel.

2.2 Secondary windows

From the main window, the user can access to additional resources:

- *Message editor* (cf. Fig. 7); this window is used to set up the score data to be sent to external devices or IanniX itself (cf. Chap. 4.6.1); every cursor or trigger is capable of sending messages defined by communication protocol, address, and a series of variables (cf. Chap. 4.1);
- *Render (performance mode)*; in this mode, the user can take advantage of dual display output, for instance by using the second output to make a video projection while keeping preview and control of the score on the main display;
- *Timer*; an additional window that displays the timecode;
- *Helper*; the help window is a useful resource for the assistance on actions performed through graphical user interface; it shows specific tips and advice as well as a list of IanniX commands corresponding to user actions;
- *Script editor*; this allows the user to edit the score file.

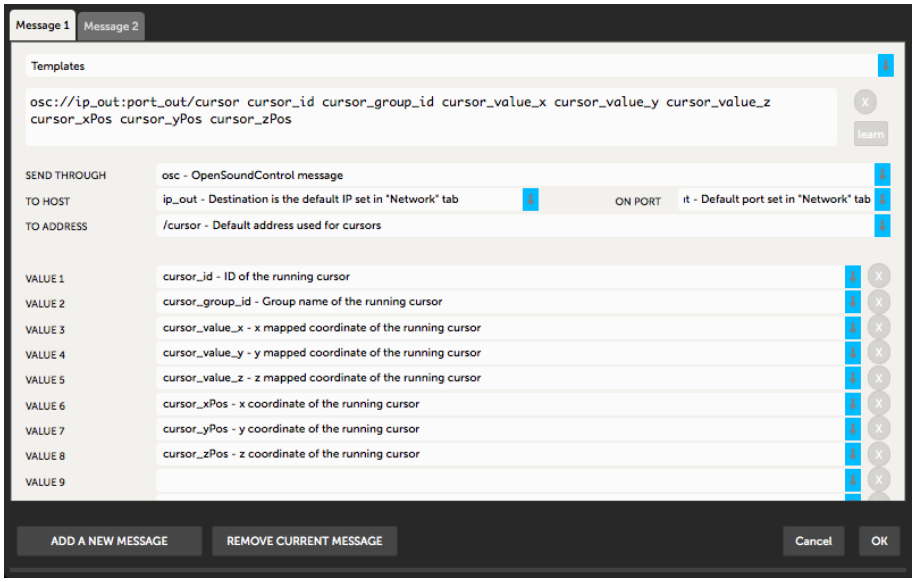

Fig. 7. Message editor.

2.2.1 Script editor

The script editor offers an advanced approach to score through JavaScript language, which underlies IanniX score files. Even with a limited knowledge of JavaScript, various features can be implemented (cf. Chap. 4.6.1). Also, the lower section of its window shows a summary of functions, commands, and variables, with a view to adding them easily in the script.

IanniX introduces a specific function for sending commands to the sequencer in order to perform an action in the score: `run()`. Commands must be provided to `run()` as a single string. General syntax is always:

```
run("<command name> <target> <arguments>");
```

For example:

```
run("setPos current 0 0 0");
```

sets the position of current object to the center of the score (X=0; Y=0; Z=0). A list of custom functions can be consulted from the JavaScript Library included in the software package (cf. Annex 1).

Commands are described extensively in Chap. 4.2; still, users can learn them in a practical manner by performing actions through graphical user interface and then finding the corresponding syntax from the *Helper*.

Possible targets are: an object ID (number), a group ID (string name of the object group), `all` (all objects), `current` (last used ID), and `lastCurve` (last used curve).

To combine numeric variables with text commands, the concatenation operator (+) must be used in order to produce a string. For example:

```
run("setPos current " + x_value + " " + y_value + " 0");
```

sets the position of current object according to user-defined variables (*value_x* and *value_y*).

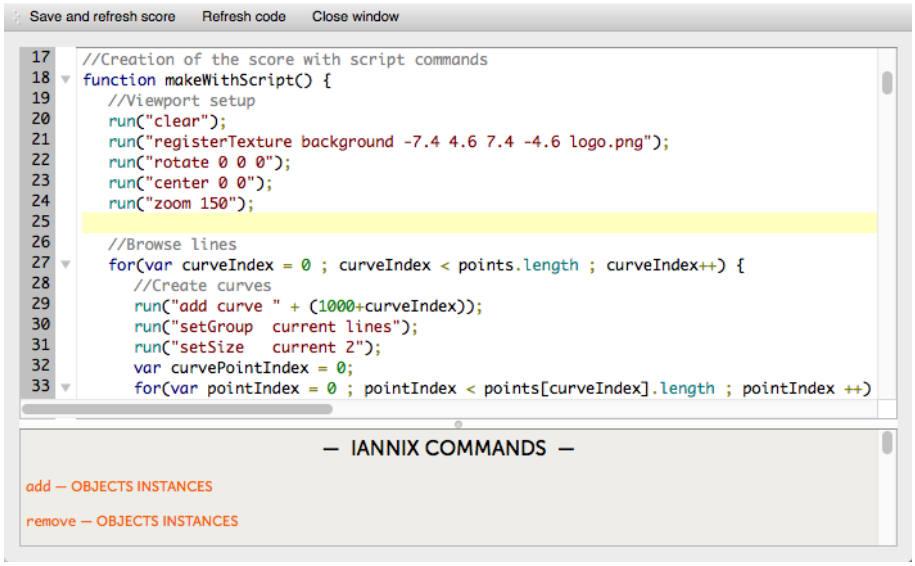

Fig. 8. Script editor.

Overall, a score file usually comprises six sections defined by JavaScript methods:

- `askUserForParameters()`; this method is called first, as it permits the user to set global variables for the subsequent generation of the score through script; the syntax is:

```
title("<displayed box title>");

ask("<menu label>", "<parameter label>",
"<parameter name>", <default value>);
```

- `makeWithScript()`; this core section is reserved to user input of code for the generation of the score (cf. Fig. 8); actions and operations are performed at the opening of a IanniX file;
- `onIncomingMessage(protocol, host, port, destination, values)`; this method is called when an incoming message is received; it is used to correlate scripts with specific input messages; for example, the position on X axis of an object (ID=3) can be adjusted by an external device that communicates through OSC protocol (cf. Chap 5.3.1):

```
if((protocol == "osc") && (destination ==
"/1/fader1")) {
var x_position = parseFloat(values[0]);
run("setPos 3 " + x_position + " 0 0"); }
```

- `madeThroughGUI()`; this method stores all actions made through graphical user interface; users should not edit this section, as it is automatically overwritten when score is saved;
- `madeThroughInterfaces()`; this method stores the actions

made by third-party devices through compatible interfaces; even in this case, users are not allowed to make changes from the script editor;

- `alterateWithScript();` this method is called last for enabling the user to add scripts with highest priority; it can be used to modify an hand-drawn score or to remove changes added accidentally by external commands.

3. IanniX objects

IanniX scores are built combining three types of abstract objects – curves, cursors, and triggers – in a three-dimensional representation system. Objects can be arranged through IanniX GUI (cf. Chap. 2.1.2), JavaScript (cf. Chap. 2.2.1), or commands received from third-party applications that use a compatible network protocol (cf. Chap. 4.3). Changes can be made in the score also during the performance.

In the definition of a score, every IanniX object has a series of general attributes that are customizable by the user:

- *ID*; an identification number used to send messages and receive commands uniquely;
- *group ID*; a string suitable to control various objects together;
- *activation status*; it determines whether an object is enabled or disabled in the score; when an object is disabled, it can not send any messages;
- *thickness / size*; attribute is defined by a floating-point number;
- *label*; a string used mainly for visualization purposes;
- *color*; it can be chosen from a standard or custom color palette;
- *texture* (only for cursors and triggers); external graphics can be imported in order to change object's default appearance;
- *messages* (only for cursors and triggers); single or multiple output messages can be set up (cf. Chap. 4.1);
- *3D position* (only for curves and triggers); it sets the object's position in the score according to X, Y, and Z axes; values are expressed in seconds [s] in order to establish a correspondence with the sequencing time of cursors.

In addition, every object type holds specific properties, functions, and attributes which are described in the chapters below.

3.1 Curves

In IanniX vocabulary, a curve is a spatial trajectory defined by a set of points which establish a sequence of sections joined together (Coduys, Jacquemin and Ranc, 2012). Each section can take three forms: a line segment (cf. Fig. 9-A), a cubic Bézier curve (cf. Fig. 9-B), or an ellipse (cf. Fig. 9-C). Otherwise, curves can be generated by mathematical equations (cf. Fig. 9-D).

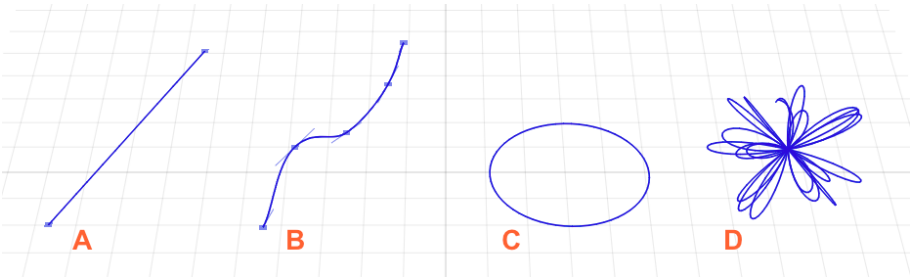

Fig. 9. Curve types: straight (A), smooth (B), circular (C), and parametric (D).

As vectors, IanniX curves have certain properties such as being interpolable, scalable, and resampleable. Operations on curves and their single points can be made from *Inspector* / *INFOS* / *3D Space* or with a direct manipulation of the score (cf. Chap. 2.1.2) as well as through action-equivalent scripts (cf. Chap. 4.2).

Curves can assume different functions in a score; if linked to one or more cursors, they constitute their support by defining the cursor's path; while in other circumstances, such as in *reactive scores* (cf. Chap. 1.2), curves might simply represent a graphical artifact. When curves intersect a cursor linked to another curve, they can describe the temporal evolution of values read by such cursor. Indeed, unlike cursors and triggers, curves do not output any message.

3.2 Cursors

While the *Transport* panel acts as global sequencer (cf. Chap. 2.1.3), IanniX cursors perform local sequencing functions. They can be considered as read heads of specific triggers and curve values. For this reason they represent the core of the poly-temporal feature.

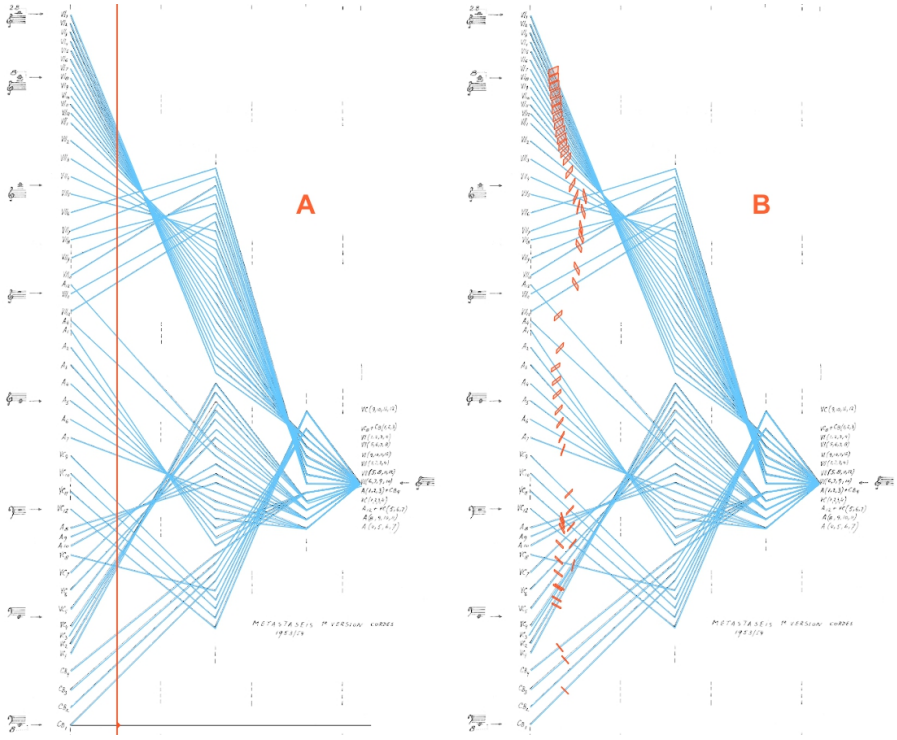

Fig. 10. Cursors in Xenakis's *Metastaseis*: mono-temporality (A) and poly-temporality (B).

Implications of poly-temporality involve specific space-time relations, as demonstrated by the score implementation of Xenakis's *Metastaseis* (cf. Fig. 10), which is part of IanniX examples.

By moving along curves according to time, cursors are capable to output a set of continuous messages related to a three-dimensional portion of the score included in their range, which is defined in terms of width and depth. For example, they can report their position over time as well as values related to colliding curves and triggers (cf. Chap. 4.1). The amount of messages sent during time can be

configured from *Inspector* / *INFOS* / *Messages*. Moreover, when a trigger is found in their field of action, cursors have the function to fire it and make it produce any instant messages (cf. Fig. 12).

An important aspect to consider when cursors send positioning values is the coordinate mapping. Included in *Messages* tab, this feature allows the user to rescale the virtual space of the score to the desired range of values needed by the receiver for controlling a specific parameter (e.g. sound frequency or amplitude). X, Y, and Z values can be mapped according to four modes:

Fig. 11. Example of coordinate mapping.

- from 0 to 1 on the bounding rectangle of the curve where the cursor moves on (cf. Fig. 11);
- from 0 to 1 on the bounding rectangle of the support curve including cursor size;
- from 0 to 1 on global score bounding rectangle;
- user-defined custom mapping.

Cursor movement is subject to global transport controls. However, cursors have several specific attributes that define their own behavior in the score:

- *cursor width* [s]; this attribute contributes to the size of the bi-dimensional action field; values are measured according to the reference of time grid (cf. Chap. 2.1.1);

- *cursor depth* [s]; it extends the action field to Z axis;
- *cursor speed / length*; the former is a factor related to global playback speed, while the latter permits to assign a fixed value of duration [s] to the cursor's path along its support curve;
- *cursor master speed*; it applies a multiplication factor to the attribute above; default value is 1, while 0 is used to stop the cursor;
- *loop pattern*; it can be one run (1 0), loop (1), single round trip (1 -1 0), or loop round trip on curve (1 -1);
- *easing curve*; it manages cursor acceleration on the support curve; the acceleration pattern can be chosen from 44 presets.
- *offset* [s]; this attribute is used to edit loop interval and starting position; negative values introduce a delay before start.

3.3 Triggers

In their default appearance, triggers are spherical objects with the capability to send single messages when are fired by a cursor. In a sense, they can be compared to musical notes, as they involve discrete events in time.

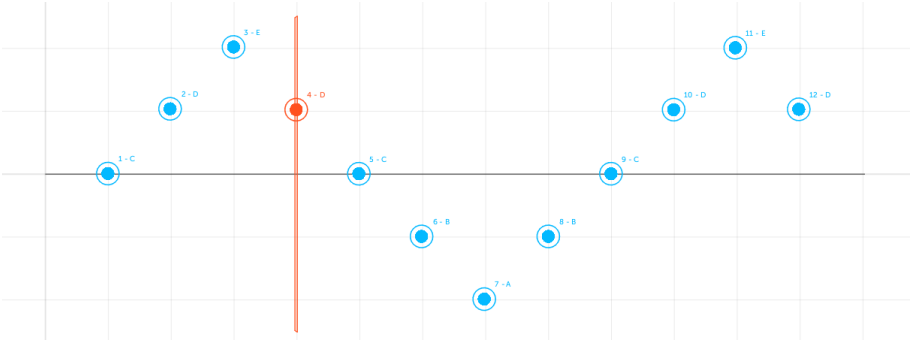

Fig. 12. Trigger fired by a cursor moving on a timeline.

For example, triggers can produce MIDI note messages in order to play a sequence of sounds on an external MIDI device (cf. Chap. 4.4).

In this case, the trigger-specific *duration* [s] attribute should be set for each note according to needs. However, triggers are able to control any sort of event, from operations on data flow to final media, depending on the software or hardware linked to IanniX. In particular, three-dimensionality of space may lead to unusual ways of conceiving a score.

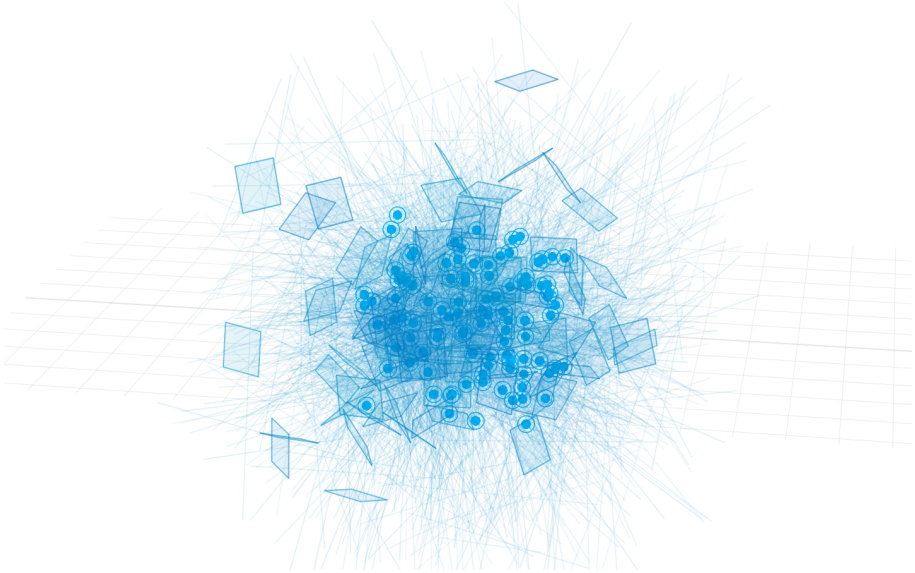

Fig. 13. Triggers, cursors, and curves in a three-dimensional score example.

4. Control and communication with other devices

IanniX operation is based on the reception of commands for the creation and the management of score data as well as on the transmission of score-related messages for controlling a third-party device. IanniX software is not intended for a self-sufficient and exclusive use. Therefore, it implements various communication protocols in order to support the interfacing with a wide range of software and hardware.

In the following chapters, IanniX variables and commands with their proper syntax will be listed with the aim of constituting a user reference on the type of data that can be extracted from scores and the actions that IanniX can carry out. Methods of interaction and respective settings will be examined, also with supporting examples.

4.1 Sent messages

As stated in the previous chapters, IanniX cursors and triggers are responsible for sending messages to other devices. The amount and the type of transmitted messages should be customized in each case by the user according to needs, for avoiding the transmission of useless data. In addition to this, general messages related to *Transport* (cf. Chap. 2.1.3) are sent directly by the sequencer.

In IanniX scripting language, every message is defined by a protocol and a possible address, followed by a series of separate variables or custom values (cf. Fig. 7). In general:

```
<protocol>://<address (if any)> <variable or value 1>
   <variable or value 2> <variable or value 3> …
```

For example:

```
osc://ip_out:port_out/cursor cursor_id cursor_xPos
        cursor_yPos cursor_zPos 0.1 abc
```

Seven communication protocols are supported for data transmission. In a message, they are declared as follows:

- `osc` - OpenSoundControl message
- `direct` - IanniX recursive/loopback message
- `midi` - MIDI message
- `serial` - ASCII string through serial port connectivity
- `http` - HTTP request to a web page or service
- `udp` - Raw UDP message
- `tcp` - XML over TCP message.

4.1.1 Cursor-related variables

Cursor-related variables can be combined in a functional way to return the temporal evolution of values and continuous magnitudes (e.g. glissando, speed, and spatial trajectory). Overall, they are:

- `cursor_id` - ID of the running cursor;
- `cursor_group_id` - group name of the running cursor;
- `cursor_label` - label of the running cursor;
- `cursor_xPos` - X coordinate of the running cursor;
- `cursor_yPos` - Y coordinate of the running cursor;
- `cursor_zPos` - Z coordinate of the running cursor;
- `cursor_value_x` - mapped X coordinate of the running cursor;
- `cursor_value_y` - mapped Y coordinate of the running cursor;
- `cursor_value_z` - mapped Z coordinate of the running cursor;

- `cursor_time` - progression of the cursor on support curve [s];
- `cursor_time_percent` - progression of the cursor from 0 to 1;
- `cursor_angle` - incidence angle of the running cursor;
- `cursor_xPos_delta` - X coordinate variation;
- `cursor_yPos_delta` - Y coordinate variation;
- `cursor_zPos_delta` - Z coordinate variation;
- `cursor_value_x_delta` - mapped X coordinate variation;
- `cursor_value_y_delta` - mapped Y coordinate variation;
- `cursor_value_z_delta` - mapped Z coordinate variation;
- `cursor_time_delta` - cursor time variation [s];
- `cursor_time_percent_delta` - cursor time variation from 0 to 1;
- `cursor_angle_delta` - incidence angle variation;
- `cursor_nb_loop` - loop counter on support curve;
- `cursor_message_ID` - message counter for cursors;
- `curve_ID` - ID of the support curve;
- `curve_group_id` - group name of the support curve;
- `curve_label` - label of the support curve;
- `curve_xPos` - X coordinate of the support curve;
- `curve_yPos` - Y coordinate of the support curve;
- `curve_zPos` - Z coordinate of the support curve;
- `collision_curve_ID` - ID of the collided curve;
- `collision_curve_group_id` - group name of the collided curve;
- `collision_curve_label` - label of the collided curve;
- `collision_curve_xPos` - X coordinate of the collided curve;
- `collision_curve_yPos` - Y coordinate of the collided curve;
- `collision_curve_zPos` - Z coordinate of the collided curve;
- `collision_xPos` - X coordinate of the collision with a curve;
- `collision_yPos` - Y coordinate of the collision with a curve;
- `collision_zPos` - Z coordinate of the collision with a curve;
- `collision_value_x` - mapped X coordinate of the collision;

- `collision_value_y` - mapped Y coordinate of the collision;
- `collision_value_z` - mapped Z coordinate of the collision;
- `collision_distance` - distance between cursor and collision.

4.1.2 Trigger-related variables

Trigger-related variables are commonly used in a message to start events on third-party devices and to control discrete data (e.g. loading a preset, or sending MIDI note and velocity). These are:

- `trigger_id` - ID of the trigger;
- `trigger_group_id` - group name of the trigger;
- `trigger_label` - label of the trigger;
- `trigger_xPos` - X coordinate of the trigger;
- `trigger_yPos` - Y coordinate of the trigger;
- `trigger_zPos` - Z coordinate of the trigger;
- `trigger_value_x` - cursor-mapped X coordinate of the trigger;
- `trigger_value_y` - cursor-mapped Y coordinate of the trigger;
- `trigger_value_z` - cursor-mapped Z coordinate of the trigger;
- `trigger_value` - trigger status (0 or 127) according to duration;
- `trigger_duration` - trigger duration [s];
- `trigger_distance` - distance between trigger and cursor;
- `trigger_side` - direction in which the trigger is hit (0 or 1);
- `trigger_message_ID` - message counter for triggers.

Triggers can also report a variable related to cursors and *vice versa*. But while the former send only one message when hit, running cursors keep sending messages over time and might return unwanted trigger values.

4.1.3 *Transport* variables

Transport variables are useful for synchronization purposes during the communication with other software (e.g. programming environments and sequencers):

- `timetag` - OSC time tag (cf. Chap. 4.3.1);
- `status` - global playback status (play, stop, or fast rewind);
- `global_time` - elapsed time [s];
- `global_time_verbose` - timecode (mmm:ss:fff).

4.2 Received commands

IanniX command messages are received from compatible network protocols as well as from recursive interface to control the whole operation of the sequencer and perform actions on IanniX scores. Likewise, they contribute to the scripting language for the creation of score files editable in the *Script editor* (cf. Chap. 2.2.1).

4.2.1 Score instances and viewport

- `load <filename>`
  loads a score file
- `clear`
  erases the content of the current score
- `registerTexture <name> <position> <filename>`
  imports an external image into the score
- `registerColor <name> <red> <green> <blue> <alpha>`
  initializes a color name from RGBA code
- `registerColorHue <name> <hue> <saturation> <value> <alpha>`
  initializes a color name from HSVA code

- `sendMessage <message>`
  sends a message in IanniX format (cf. Chap. 4.1)
- `log <text>`
  logs information to IanniX message log
- `mouse <x> <y>`
  sets mouse cursor position in the viewport
- `center <x> <y> 0`
  moves the camera position in the viewport
- `rotate <angle x> <angle y> <angle z>`
  rotates the camera position in the viewport
- `zoom <value>`
  changes the current zoom in the viewport.

4.2.2 Object instances and identification

- `add <trigger/curve/cursor> <ID/auto>`
  adds a IanniX object to the score and sets its ID
- `remove <ID>`
  removes an object from the score
- `setId <old ID> <new ID>`
  changes the object ID
- `setGroup <target> <name>`
  sets the group for one or more objects (cf. Chap. 2.2.1)
- `setLabel <target> <name>`
  sets the label for one or more objects
- `setActive <target> <0/1>`
  sets the activation status for one or more objects
- `setSolo <target> <0/1>`
  sets the *solo* mode for one or more objects
- `setMute <target> <0/1>`
  sets the *mute* mode for one or more objects

4.2.3 Object positioning

- `setPos <target> <x> <y> <z>`
  sets the absolute position of an object
- `setPosX <target> <x>`
  sets the X coordinate of an object
- `setPosY <target> <y>`
  sets the Y coordinate of an object
- `setPosZ <target> <z>`
  sets the Z coordinate of an object
- `setTranslate <target> <Δx> <Δy> <Δz>`
  shifts the position of an object
- `setTime <target> <time [s]>`
  sets the cursor position in relation to its support curve
- `setTimePercent <target> <time from 0 to 1>`
  sets the cursor position in relation to its support curve
- `setCurve <target> <curve ID/lastCurve>`
  links a cursor with a support curve
- `setPointAt <target> <index> <x> <y> <z>`
  sets the point position on a curve (straight)
- `setSmoothPointAt <target> <index> <x> <y> <z>`
  sets the point position on a curve (smooth)
- `setPointXAt <target> <index> <x>`
  sets the X coordinate of a point on the curve
- `setPointYAt <target> <index> <y>`
  sets the Y coordinate of a point on the curve
- `setPointZAt <target> <index> <z>`
  sets the Y coordinate of a point on the curve
- `removePointAt <target> <index>`
  removes a point from the curve
- `translatePoint <target> <index> <Δx> <Δy> <Δz>`
  shifts the position of a point on the curve

- `translatePoints <target> <Δx> <Δy> <Δz>`
  shifts the position of all points on the curve
- `shiftPoints <target> <index> <-1/1>`
  shifts points of a curve in a direction
- `setResize <target> <width> <height>`
  resizes a curve according to width and height
- `setResizeF <target> <scale factor>`
  resizes a curve according to a scale factor
- `displayCurveEditor <target> 1`
  shows the point editor for a curve
- `displayCurveResample <target> 1`
  shows the resampling tool for a curve
- `setPointsEllipse <target> <width> <height>`
  sets a circular curve according to its size
- `setEquation <target> cartesian <x eq.>,<y eq.>,<z eq.>`
  sets a parametric curve according to cartesian coordinates
- `setEquation <target> polar <r eq.>,<φ eq.>,<θ eq.>`
  sets a parametric curve according to polar coordinates
- `setEquationParam <target> <parameter name> <value>`
  sets a parameter value in the curve equation
- `setEquationNbPoints <target> <number of points>`
  sets the number of points to calculate the curve equation
- `setPointsTxt <target> <scale factor> <font> <text>`
  sets a curve from text characters
- `setPointsPath <target> <scale factor> <SVG path>`
  sets a curve from path data (“SVG Path”, 2017)
- `setPointsLines <target> <scale factor> <points (x,y)>`
  sets a straight curve from polyline (“SVG Polyline”, 2017).

4.2.4 Object appearance

- `setSize <target> <thickness/size>`
  sets the thickness or size of an object

- `setWidth <target> <value>`
  sets the cursor width
- `setDepth <target> <value>`
  sets the cursor depth
- `setColor <target> <name/RGBA code>`
  sets the color of an object using RGB color space
- `setColorActive <target> <name/RGBA code>`
  sets the color of an active object using RGB
- `setColorInactive <target> <name/RGBA code>`
  sets the color of an inactive object using RGB
- `setColorMultiply <target> <name/RGBA code>`
  sets the color multiplication for an object using RGB
- `setColorHue <target> <name/HSVA code>`
  sets the color of an object using HSV color space
- `setColorActiveHue <target> <name/HSVA code>`
  sets the color of an active object using HSV
- `setColorInactiveHue <target> <name/HSVA code>`
  sets the color of an inactive object using HSV
- `setColorMultiplyHue <target> <name/HSVA code>`
  sets the color multiplication for an object using HSV
- `setTexture <target> <name>`
  sets a preloaded texture for an object
- `setTextureActive <target> <name>`
  sets a preloaded texture for an active object
- `setTextureInactive <target> <name>`
  sets a preloaded texture for an inactive object.

4.2.5 Object behavior

- `setSpeed <target> <factor>`
  sets the cursor speed as a factor related to *Transport*
- `setSpeed <target> auto <path end [s]>`
  sets the duration of entire cursor path on support curve

- `setSpeedF <target> <factor>`
  applies a multiplication factor on cursor speed
- `setOffset <target> <initial time [s]> <start [s]> <end [s]/end>`
  sets the offset parameters for a cursor
- `setPattern <target> <easing> 0 <loop pattern>`
  sets loop pattern and cursor acceleration
- `setBoundsSource <target> <range x> <range y> <range z>`
  sets a custom mapping area for a cursor
- `setBoundsSourceMode <target> <0/1/2/3>`
  sets the coordinate mapping mode for a cursor
- `setBoundsTarget <target> <range x> <range y> <range z>`
  sets the range of output values for a cursor
- `setFire <target> <none/group/all>`
  sets the trigger firing mode for a cursor
- `setMessage <target> <interval [ms]>, <msg1>, <msg2>, …`
  sets one or more messages to be transmitted by an object
- `setMessageInterval <target> <interval [ms]>`
  sets the message transmission rate for an object
- `trig <target>`
  forces an object to send its preset messages
- `setTriggerOff <target> <duration [s]>`
  sets the trigger duration
- `setElasticity <target> <factor>`
  sets the elasticity factor of a curve.

4.2.6 Sequencing

- `play <speed factor (optional)>`
  starts the score playback
- `stop`
  stops the score playback
- `fastrewind`
  resets the score to initial position

- `goto <time [s]>`

  goes to a specific timecode

- `speed <factor>`

  changes the global speed factor in relation to grid.

4.3 Network protocols

IanniX is capable of sending data to and receiving data from devices connected to local or remote networks. Supported network protocols are OSC, raw UDP, TCP, and HTTP. As a consequence, this feature significantly expands software interfacing possibilities and interaction strategies.

4.3.1 OSC

Developed at the CNMAT (Center of New Music and Audio Technologies) of the University of California at Berkeley, OSC (OpenSoundControl) is a message-based protocol for the communication between computers, sound synthesizers, and other multimedia devices that is optimized for modern networking technology (Wright and Freed, 1997). OSC is currently implemented in a wide range of computer applications, including IanniX as a control-message-generating software (Wright, 2005).

OSC message packets are typically transmitted via UDP (User Datagram Protocol) to an IP address on a destination network port:

```
osc://<IP address>:<destination port>/<message>
```

For example:

```
osc://192.168.1.3:57120/transport play 1
```

Since OSC does not provide any mechanism for clock synchronization, OSC time tags are used in the transmission of OSC bundles for temporal representation.

In IanniX, OSC settings can be configured from *Inspector* / *CONFIG* / *Network*.

A convenient way to route and manage OSC messages from and to IanniX foresees the communication with a third-party programming environment. For instance, Max ("Max Software Tools for Media | Cycling '74", 2017) allows for audio/video processing by means of a programmable data flow that can be also controlled via OSC; an example is included in IanniX software package (Patches/Max/Max Sound Example.maxpat).

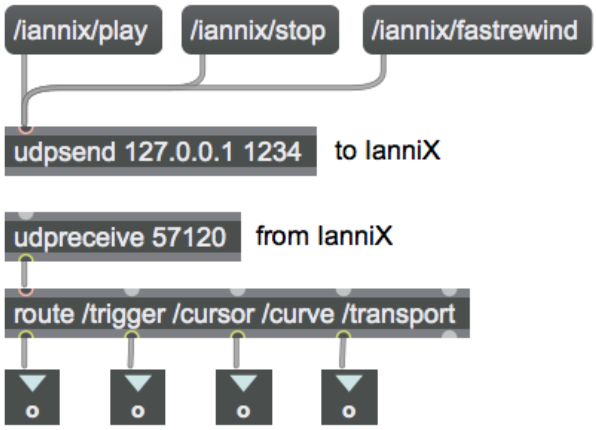

Fig. 14. OSC message routing in Max programming environment.

Communication between Max and IanniX takes places via two objects: *udpreceive* <*inbound port*> and *udpsend* <*host IP*> <*outbound port*>. Conventionally, when both applications run on the same local machine, the IP address must be set to *127.0.0.1* (or *localhost*). Otherwise, IP should be set according to network preferences.

4.3.2 Raw UDP

This protocol is used to send and receive messages through a very common method for the transport of data over the Internet. Analogously to OpenSoundControl, raw UDP packets employ a reduced data bandwidth at the expense of unreliability that characterizes unidirectional communication: when a message is sent over UDP, there is no computational way to verify whether it will be actually received (Kumar and Rai, 2012).

For instance, messages may be transmitted to control the sequencer with Pure Data programming environment ("Pure Data – Pd Community Site", 2017).

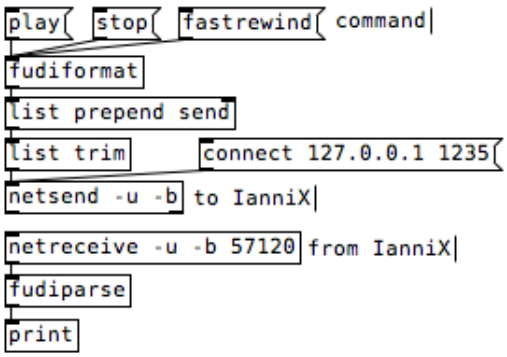

Fig. 15. Example of communication with Pure Data 0.48.

In Pure Data, *netsend* object must be initialized with a message:

```
connect <IP address> <destination port>
```

Messages should pass through *fudiformat* and *fudiparse* objects in order to be formatted correctly.

A Pure Data patch for basic communication is provided by IanniX (Patches/PureData/PureData Example.pd).

4.3.3 TCP and WebSocket

Another common network protocol which is compatible with IanniX is TCP (Transmission Control Protocol). Unlike UDP, TCP is based on a connection-oriented bidirectional communication, which is more reliable and robust in the delivery of data streams but may generate more latency (Kumar and Rai, 2012).

IanniX can act as a TCP client/server for the transfer of data in raw TCP format or XML ("XML Introduction", 2017). General syntax for outgoing messages is:

```
tcp:// <XML node 1> <XML node 2> <XML node 3> …
```

TCP port and data format are customizable from *Inspector* / *CONFIG* / *Network*.

Possible applications include operations in which accuracy is requested in data transfer. IanniX proposes an example created with Flash (Patches/Adobe Flash/Flash.fla) where cursor values may be used to create graphic animations.

Moreover, TCP-based WebSocket protocol allows for simultaneous bidirectional transmission of data exploitable for displaying and controlling IanniX interface on a web page. A simple implementation is included in the software bundle (Tools/HTML Template.html).

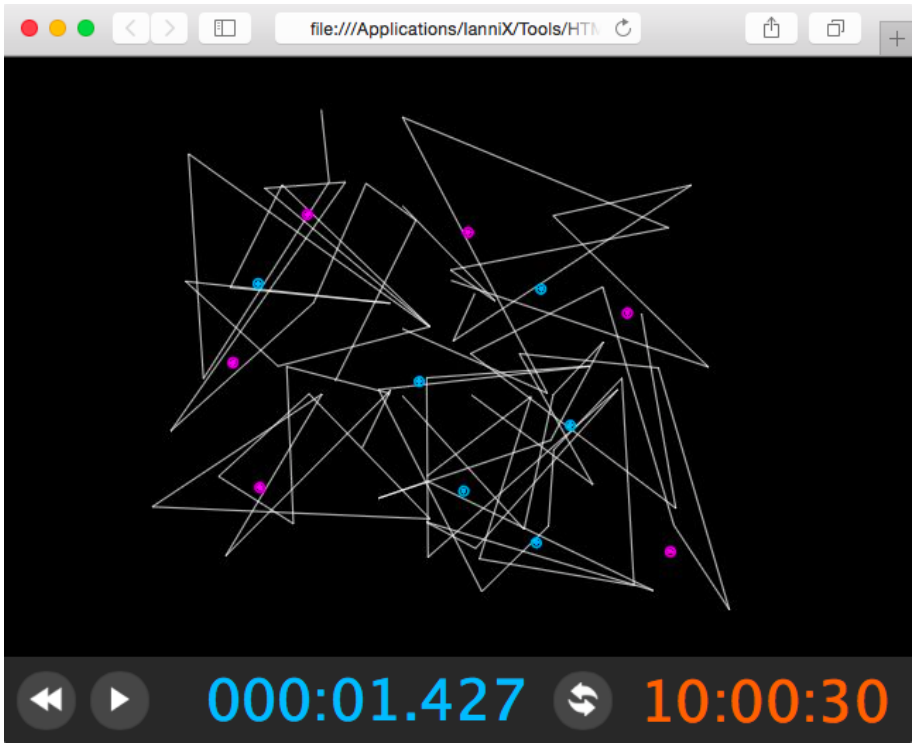

Fig. 16. IanniX interface on a web page.

4.3.4 HTTP

Based on a client-server architecture, HTTP (HyperText Transfer Protocol) is one of the main systems for transmitting information over the web.

With outgoing messages, IanniX is capable of managing HTTP requests to web pages or web services. The message syntax is:

```
http://<host IP>:<port>/<address> <query argument 1>
      <query argument 2> <query argument 3> …
```

While the embedded HTTP server accepts IanniX commands from web browsers:

```
http://<IanniX server IP>:<port>/<address (optional)>?
      =<command 1>&=<command 2>&=<command 3> …
```

For example:

```
http://127.0.0.1:1236/?=add%20cursor%20auto&=play
```

4.4 MIDI interface

MIDI (Musical Instrument Digital Interface) is a technical specification for the connection and the communication between electronic musical instruments and other devices such as sequencers, computers, lighting controllers, and mixers. Originally conceived for live performance, its developments have had "an enormous impact in recording studios, audio and video production, and composition environments" ("The Complete MIDI 1.0 Detailed Specification", 2014). Indeed, MIDI data are extremely compact and therefore suited for realtime accuracy.

IanniX integrates a virtual MIDI output device (i.e. *From IanniX*) and a MIDI input (i.e. *To IanniX*) that are accessible from software side as well as from external hardware, normally passing through a USB port.

Through MIDI output, IanniX cursors and triggers may send several types of messages that comply with MIDI specifications (Ibid.):

- *Note* messages

  `midi://midi_out/note <MIDI channel> <note number> <velocity> <duration>`
- *Control Change* messages

  `midi://midi_out/cc` `<MIDI channel> <controller number> <value>`
- *Program Change* messages

  `midi://midi_out/pgm` `<MIDI channel> <program number>`

- *Pitch Bend Change* messages

  midi://midi_out/bend <MIDI channel> <value>

A IanniX variable or a custom value can be assigned to each field (cf. Chap. 4.1), although in some cases a conversion of variable type might be needed. For example, the *label* attribute of a trigger – which normally takes a string – may accept an integer number in order to control a MIDI velocity value (from 0 to 127). In this case, outgoing message should be set as follows:

```
midi://midi_out/note <MIDI channel> <note number>
     {parseInt(trigger_label)} <duration>
```

Due to the format of MIDI messages, IanniX can not receive commands directly from an external MIDI device. However, this protocol can be still used for control purposes by implementing a custom script that generates a command or an action according to incoming messages (cf. Chap. 2.2.1).

4.5 Serial interface

This interface is oriented to the communication with prototyping kits and microcontroller boards – such as Arduino – through serial port connectivity (UART). Intended uses of IanniX serial interface include the control of actuators and the reception of sensor data. Serial port path and settings are customizable from *Inspector* / *CONFIG* / *Arduino*.

In order to be compatible with a wide number of devices, IanniX messages are encoded in standard ASCII format and separated by spaces, ending with a CR control character (“ASCII - Wikipedia, the free encyclopedia”, 2017).

Instead, for sending IanniX commands from Arduino boards, *println()* function is used ("Arduino - Println", 2017). For instance:

```
Serial.println("zoom 100");
```

An exemplification for generating tones and receiving messages is provided (Examples/Simple Arduino example.iannix and Patches/Arduino/Arduino.ino).

4.6 Software interfaces

4.6.1 Recursivity

IanniX implements an *ad-hoc* protocol for sending commands to the software itself through the output of messages. In this way, the output of the score is looped to its input.

The syntax of outbound messages may include both variables and custom values as attributes for setting a command (cf. Chap. 4.2):

```
direct:// <command> <variables and custom values>
```

For example:

```
direct:// setPos 3 {cursor_xPos+1} 1 0
```

Through *direct* messages, objects in a score are capable of producing various actions and space-time effects according to the type of recursive algorithm (Coduys and Jacquemin, 2014). The following scripts demonstrate a few significant cases:

- control of the sequencer through a IanniX object

```
run("add curve 1");
run("setPointAt lastCurve 0 0 0");
```

```
run("setPointAt lastCurve 1 5 0");
run("add cursor 2");
run("setCurve current lastCurve");
run("add trigger 3");
run("setPos 3 5 0 0");
run("setMessage 3 direct:// stop"); //stop the sequencer
```

- control of an object by another object

```
run("add curve 1");
run("setPointsLines lastCurve 1 0,0 3,1 5,0");
run("add cursor 2");
run("setCurve current lastCurve");
run("add trigger 3");
run("setPos current 1 1 0");
run("setMessage 2 direct:// setPos 3 {cursor_xPos+1}
{cursor_yPos+1} 0"); //trigger follows cursor position
```

- stable recursive setting

```
var iMax = 2;
for(var i = 0 ; i < iMax ; i++) {
run("add curve " + (100+i));
run("setPos current 3 3 0");
run("setEquation current polar radius, TWO_PI*t, theta");
run("setEquationParam current radius " + (1+i));
run("setEquationParam current theta 0");
run("add cursor " + i);
run("setSpeed current lock " + map(i, 0, iMax, 0.2, 0.3));
run("setCurve current lastCurve");
run("setPattern current 0 0 1"); //loop
}
run("setMessage 0 5, direct:// setEquationParam 101 radius
cursor_yPos"); //cursor 0 sets curve 101 radius
run("setMessage 1 5, direct:// setEquationParam 100 radius
cursor_xPos"); //cursor 1 sets curve 100 radius
```

- continue amplification or progressive extinction

```
run("add curve 1");
run("setPointAt lastCurve 0 0 0");
run("setPointAt lastCurve 1 1 1");
run("add cursor 2");
run("setCurve current lastCurve");
run("setMessage current direct:// setPointAt lastCurve 1 1
{cursor_yPos+1}"); //move point according to cursor pos.
```

- chaotic behavior

```
run("add curve 1");
run("setPointsEllipse lastCurve 1 1"); //circle radius = 1
run("add cursor 2");
```

```
run("setCurve current lastCurve");
run("setPattern current 0 0 1"); //loop
run("setMessage current direct:// setResize lastCurve
{cursor_xPos+1} {cursor_yPos+1}"); //cursor position shapes
the ellipse
```

4.6.2 Syphon

Syphon is an open-source software technology for sharing video and still frames among applications running in realtime ("Syphon", 2017).

IanniX integrates a Syphon output that replicates the score visualization window. This function is accessible from *Inspector* / *CONFIG* / *Software.*

Through this interface, the performance of IanniX scores can be imported as video input on compatible third-party software. More and more applications support Syphon, such as Max, MadMapper, and Syphon Recorder. Possible usages include video processing in realtime, projection mapping, and video recording.

In addition, since version 0.9.17 IanniX supports Syphon input for adding custom textures to cursors and triggers (cf. Chap. 3).

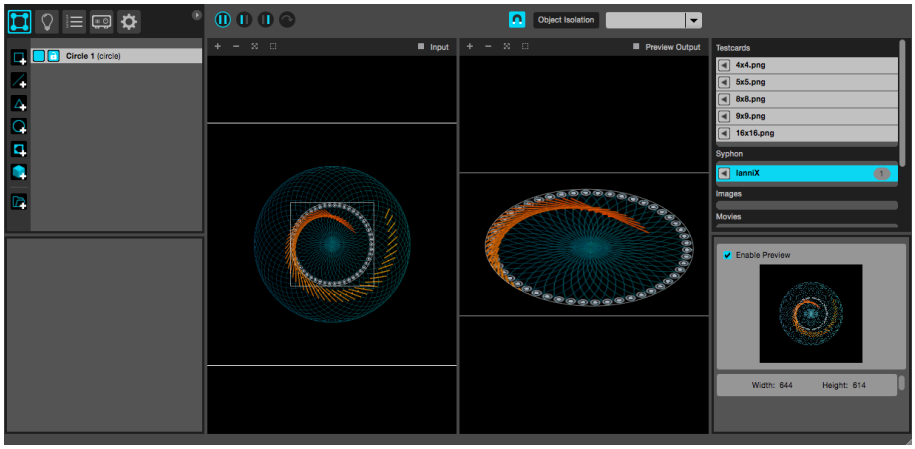

Fig. 17. IanniX score imported into MadMapper while running.

References

Alessandretti, S. & Sparano, G. (2014). NEYMA, interactive soundscape composition based on a low budget motion capture system. *Proceedings of the International Computer Music Conference | Sound and Music Computing Conference* (pp. 379-383). Athens, Greece: ICMA.

Arduino (2017). *Arduino - Println*. Retrieved October 30, 2017 from https://www.arduino.cc/en/Serial/Println

Bellona, J. (2013). *Sonification Study of San Giovanni Elemosinario*. Retrieved March 14, 2016 from http://jpbellona.com/public/writing/BellonaJon_SanGiovanni.pdf

Bourotte, R. (2012). The UPIC and its descendants: drawing sound 2012. *Proceedings of the Symposium Xenakis. La musique électroacoustique*. Retrieved March 14, 2016 from http://www.cdmc.asso.fr/sites/default/files/texte/pdf/rencontres/intervention18_xenakis_electroacoustique.pdf

Cycling '74 (2017). *Max Software Tools for Media | Cycling '74*. Retrieved October 17, 2017 from https://cycling74.com/products/max/

Coduys, T. & Ferry, G. (2004). IanniX. Aesthetical/Symbolic visualisations for hypermedia composition. *Proceedings of the Sound and Music Computing Conference*. Retrieved February 16, 2016 from http://smcnetwork.org/files/proceedings/2004/P18.pdf

Coduys, T. & Jacquemin, G. (2014). Partitions retroactives avec IanniX. *Actes des Journées d'Informatique Musical*e. Retrieved February 17, 2016 from http://jim.afim-asso.org/jim2014/images/0040_01_03_PARTITIONS%20RETROACTIVES%20AVEC%20IANNIX.pdf

Coduys, T., Jacquemin, G. & Ranc, M. (2012). IanniX 0.8. *Actes des Journées d'Informatique Musical*e. Retrieved February 17, 2016 from http://jim.afim-asso.org/jim12/pdf/jim2012_18_p_jacquemin.pdf

Coduys, T., Lefèvre, A. & Pape, G. (2003). IanniX. *Actes des Journées d'Informatique Musical*e. Retrieved February 17, 2016 from http://jim.afim-asso.org/jim2003/articles/iannix.pdf

IanniX Association (2012). *World Expo 2012 Korea | IanniX*. Retrieved October 17, 2017 from https://www.iannix.org/en/item/world-expo-2012-korea/

IanniX Association (2017). *Showcase | IanniX*. Retrieved October 17, 2017 from https://www.iannix.org/en/projects/

JavaScript (2016). *JavaScript | MDN*. Retrieved March 17, 2016 from https://developer.mozilla.org/en-US/docs/Web/JavaScript

Kumar, S. & Rai, S. (2012). Survey on Transport Layer Protocols: TCP & UDP. *International Journal of Computer Applications*, 46 (7), 20-25.

Manaris, B. & Stoudenmier, S. (2015). Specter: Combining Music Information Retrieval with Sound Spatialization. *Proceedings of the International Society for Music Information Retrieval Conference.* Retrieved March 15, 2016 from http://ismir2015.uma.es/articles/270_Paper.pdf

MIDI Association (2014). T*he Complete MIDI 1.0 Detailed Specification*. Retrieved October 25, 2017 from https://www.midi.org/specifications/item/the-midi-1-0-specification

Pure Data (2017). Pure Data – Pd Community Site. Retrieved October 17, 2017 from https://puredata.info

Raczinski, J.M., Marino, G. & Serra, M.H. (1990). The new UPIC system. *Proceedings of the International Computer Music Conference.* Retrieved February 16, 2016 from http://quod.lib.umich.edu/i/icmc/bbp2372.1990.070/1

Ranc, M. (2013). *Du temps à l'espace. De l'utilisation d'un séquenceur graphique comme outil d'architecture sonore*. Master's Thesis in Innovation by Design, ENSCI–Les Ateliers, Paris. Retrieved October 25, 2017 from https://issuu.com/ensci-design/docs/memoire_matthieu_ranc

Scordato, J. (2015). Vision II: an audiovisual performance with IanniX. *Living Lab Music 5 + DI_Stanze. Atti/Proceedings*. Retrieved February 16, 2016 from https://issuu.com/sampl-lab/docs/living_lab_music_5

Syphon (2017) *Syphon*. Retrieved October 17, 2017 from http://syphon.v002.info

W3Schools (2017). *SVG Path*. Retrieved October 25, 2017 from https://www.w3schools.com/svg/svg_path.asp

W3Schools (2017). *SVG Polyline*. Retrieved October 25, 2017 from https://www.w3schools.com/graphics/svg_polyline.asp

W3Schools (2017). *XML Introduction*. Retrieved October 25, 2017 from https://www.w3schools.com/xml/xml_whatis.asp

Wikipedia (2016). *ASCII - Wikipedia, the free encyclopedia*. Retrieved October 25, 2017 from https://en.wikipedia.org/wiki/ASCII

Wright, M. (2005). Open Sound Control. An enabling technology for musical networking. *Organised Sound*, 10 (3), 193-200.

Wright, M. & Freed, A. (1997). Open SoundControl. A New Protocol for Communicating with Sound Synthesizers. *Proceedings of the International Computer Music Conference.* Retrieved April 17, 2016 from http://quod.lib.umich.edu/i/icmc/bbp2372.1997.033/1

Annex 1: JavaScript Library

```
/*
    This file is part of IanniX, a graphical real-time open-source sequencer for
digital art
    Copyright (C) 2010-2015 — IanniX Association

    Project Manager: Thierry Coduys (http://www.le-hub.org)
    Development:     Guillaume Jacquemin (http://www.buzzinglight.com)

    This file was written by Guillaume Jacquemin.

    IanniX is a free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/


//Shortcut to IanniX Object
function run(_command)   {
        return iannix.execute(_command);
}
function ask(_category, _label, _variable, _defaultValue) {
        return iannix.ask(_category, _label, _variable, _defaultValue);
}
function title(_title)   {
        return iannix.meta(_title);
}
function console(_log)   {
        return iannix.execute("log " + JSON.stringify(_log, null, 4));
}
function load(_filename) {
        return iannix.load(_filename);
}
function loadJSON(_filename) {
        return JSON.parse(load(_filename));
}

//Prototypes for Strings
String.prototype.trim = function()
        { return (this.replace(/^[\s\xA0]+/, "").replace(/[\s\xA0]+$/, "")) }
String.prototype.startsWith = function(str)
        { return (this.match("^"+str)==str) }
String.prototype.endsWith = function(str)
        { return (this.match(str+"$")==str) }
String.prototype.replaceAll = function(str, str2)
```

```
        { return (this.replace(new RegExp(str, 'g'), str2)) }
String.prototype.pad = function(length) {
        var str = '' + this;
    while (str.length < length) {
        str = '0' + str;
    }
    return str;
}


//Constants
var E          = Math.E;
var LN2        = Math.LN2;
var LN10       = Math.LN10;
var LOG2E      = Math.LOG2E;
var LOG10E     = Math.LOG10E;
var PI         = Math.PI;
var TWO_PI     = 2 * Math.PI;
var THIRD_PI   = Math.PI / 3;
var QUARTER_PI = Math.PI / 4;
var HALF_PI    = Math.PI / 2;
var SQRT1_2    = Math.SQRT1_2;
var SQRT2      = Math.SQRT2;

//Math functions
function abs(x)                 {       return Math.abs(x);       }
function acos(x)                {       return Math.acos(x);      }
function asin(x)                {       return Math.asin(x);      }
function atan(x)                {       return Math.atan(x);      }
function atan2(x,y)             {       return Math.atan2(x,y);   }
function ceil(x)                {       return Math.ceil(x);      }
function cos(x)                 {       return Math.cos(x);       }
function exp(x)                 {       return Math.exp(x);       }
function floor(x)               {       return Math.floor(x);     }
function log(x)                 {       return Math.log(x);       }
function min(x,y)               {       return Math.min(x,y);     }
function max(x,y)               {       return Math.max(x,y);     }
function pow(x,y)               {       return Math.pow(x,y);     }
function sin(x)                 {       return Math.sin(x);       }
function sqrt(x)                {       return Math.sqrt(x);      }
function sq(x)                  {       return x*x;               }
function tan(x)                 {       return Math.tan(x);       }
function degrees(value)         {       return value * 180. / pi;  }
function radians(value)         {       return value * pi / 180.;  }
function round(x, y)            {
  if(y == undefined)     return Math.round(x);
  else                   return Math.round(x*Math.pow(10, y)) / Math.pow(10, y);
}
function random(low, high)      {
        if((low == undefined) || (high == undefined))
                return Math.random();
        else
                return range(Math.random(), low, high);
}

//Useful functions
function constrain(value, min, max) {
        return Math.min(max, Math.max(min, value));
}
```

```
function dist(x1, y1, z1, x2, y2, z2) {
        var dx = x2 - x1, dy = y2 - y1, dz = z2 - z1;
        return Math.sqrt(sq(dx) + sq(dy) + sq(dz));
}
function angle(x1, y1, x2, y2) {
        var dx = x2 - x1, dy = y2 - y1, angle = 0;
        if((dx > 0) && (dy >= 0))
            angle = (Math.atan(dy / dx)) * 180.0 / PI;
        else if((dx <= 0) && (dy > 0))
            angle = (-Math.atan(dx / dy) + HALF_PI) * 180.0 / PI;
        else if((dx < 0) && (dy <= 0))
            angle = (Math.atan(dy / dx) + PI) * 180.0 / PI;
        else if((dx >= 0) && (dy < 0))
            angle = (-Math.atan(dx / dy) + 3 * HALF_PI) * 180.0 / PI;
        return angle;
}
function norm(value, low, high, exp) {
        if((high - low) == 0)
                return 0;
        else
                return linexp((value - low) / (high - low), exp);
}
function range(value, low, high, exp) {
        value = linexp(value, exp);
        return value * (high - low) + low;
}
function rangeMid(value, low, mid, high, exp) {
        value = linexp(value, exp);
        if(value < 0.5)
                return (value * 2) * (mid - low) + low;
        else
                return (value - .5) * 2 * (high - mid) + mid;
}
function map(value, low1, high1, low2, high2, exp) {
        return range(norm(value, low1, high1, exp), low2, high2);
}

function linexp(value, factor) {
        if((factor == undefined) || (factor == 0))
                return value;
        else
                return (exp(factor * value - factor) - exp(-factor)) / (1 -
exp(-factor));
}
```