SYMBOLIST RE-IMAGINED: DEVELOPING TOOLS FOR USER-DEFINED BIDIRECTIONAL MAPPING

Rama Gottfried
Hochschule für Music und Theater
Hamburg, Germany
rama.gottfried@hfmt-hamburg.de

ABSTRACT

maybe already start with history here, and then continue to symbolist?

SYMBOLIST is an in-development application for experimental notation, with the goal of creating a working environment for developing symbolic notation for multimedia which can be interpreted and performed by electronics. The program aims to provide an open play space, with tools for experimentation, and thinking visually about relationships between representation and interpretation in media performance. In the paper we discuss the evaluation and re-design of the application based on the need for a bidirectional mapping framework for working with symbolic notation and its corresponding data representations.

1. BACKGROUND

The origins of the SYMBOLIST project can be traced back to 2011 when I was working on my PhD at UC Berkeley's Center for New Music and Audio Technologies (CNMAT). Around this time, I began composing in Adobe Illustrator using a plugin called Scriptographer [1], which allowed the user to create new drawing tools with Javascript that could then be used like a brush in Illustrator; much like you would use mouse interaction in programs like Processing [2]. This was perfect for my composition needs working with extended instrumental techniques, since I could then create a notation of whatever I needed and code it into a reusable graphic template, that could then be manipulated graphically in Illustrator. Additionally, since you had access to the mouse movement, you could create interactions that could be used to compose, for instance I often used what I called a "notehead-line", which was a note-head of some shape, with a line extending out from it to show its duration. Using Scriptographer, I was able to create a user interface where after clicking down on the Illustrator canvas to draw the note-head, I could then track the mouse drag to determine the end of the duration line.

Scriptographer's drawing API provided a set of basic vector primitives and grouping methods that closely parallel the objects found in Scalable Vector Graphics (SVG) format [3] which is well supported in Adobe Illustrator. Using grouping methods in Scriptographer, you could create hierarchies of objects that would then be created in Illustrator as grouped objects visible as nested folders in the layers menu.

Around the same time at CNMAT, I was deeply involved with developing approaches to instrument design using Open Sound Control (OSC) [1] as a principle data structuring encoding. One day, while working with Scriptographer, I had saved a score in Illustrator as SVG format and accidentally opened the SVG file in a text editor. In the SVG file I noticed that all of my graphic objects where there in a human readable format, and closely resembled the kind of nested objects that we were working on at CNMAT in the Odot library[2]. This gave me the idea that I could maybe translate the SVG information into OSC and then "perform" the OSC score in much the same way as you would a stream of OSC coming from a sensor based instrument. The first tests seemed promising, and so I logged it away for possible future use.

Soon after, studying high-resolution spatial audio rendering systems, I needed to find a way to compose spatial movements in a way that would connect with my graphic compositional practice. After some initial experiments using Blender [4] to compose movements which could then be parsed via Python and sent out over OSC, I was confronted by a perceptual gap between common practice notation and 3D representation, which seemed too difficult to address in the limited time I had, and so I fell back on using automation controls in Ableton Live [5] and sending OSC to control the movements using Max for Live [6]. This was functional, but had the limitation of forcing the composer to separate each data parameter into separate streams of data, whereas in a symbolic representation different graphic attributes can indicate different elements of the data at the same time. Following these experiences [3] I later returned to the SVG-OSC transcoding idea, and produced the first working model which was presented at the 2015 TENOR conference [4].

In the meantime, the Scriptographer project was abandoned by its developers after Adobe drastically changed their plugin API in version CS6. Rather than recode the project from scratch, the authors went on to create Pa-

[1] https://scriptographer.org/
[2] https://processing.org/

[3] https://www.w3.org/TR/SVG11
[4] https://www.blender.org/
[5] https://www.ableton.com/en/
[6] https://cycling74.com/

per.js [7], which has some similarities with Scriptographer, but is more closely related to Processing, since it no longer is bound to the Illustrator application environment. This created a stumbling block for the SVG-OSC work, since I was relying on Scriptographer and its link to Illustrator as primary tools for the creation of easy to print scores for musicians to perform with. Additionally, working with preliminary OSC transcoding tests I found that it was becoming somewhat complicated to parse complex hierarchies of symbols in order to play them back via OSC streams, and so in 2017 I began work in collaboration with OpenMusic developer Jean Bresson [5], through a Ircam-ZKM Musical Research Residency towards the goal of creating a system that could replace the Scriptographer/Illustrator approach that I had developed so far.

Symbolist JUCE

The first version of SYMBOLIST was created in 2018 as a standalone JUCE [8] application, which provided the basic tools for drawing vector graphics, and query system that allowed SYMBOLIST to be used as a lookup table for OSC stream playback [6].

Working in JUCE seemed practical since it has a wide user base, and is used for audio plugins as well as Max and Ableton Live applications. However, working with SVG in JUCE proved to be an issue, since JUCE does not support the full SVG specification [9]. However, a greater turning point in the project occurred towards the end of residency as I was thinking about how to simplify the process of using the SVG data for controlling digital processes.

Clefs and Bidirectional Mapping

In the first version of SYMBOLIST as in the original SVG-OSC implementation, the graphic data needed to be interpreted by the application that received the data. Using Odot I would parse the graphic OSC information coming in from SYMBOLIST, and then mapped the data to other processes, for instance synthesis parameters, or coordinates for spatial rendering. This process of interpretation requires that the interpreter know the context of the graphic objects. For example, that a circle is a note-head and not a rhythmic dot, and so on. Like keys on the axises of a graphic plot of information, musical symbols, meter, stafflines and clefs, indicate to the reader how they should interpret the notes and rhythmic symbols written on the staff.

The following phase of SYMBOLIST development continued as I began work at the Hamburg University of Music and Theater, working with Georg Hajdu in the Innovative Hochschule project. Continuing on this idea of the *clef* as being a plot "key" for interpretation, I began work on developing a system for SYMBOLIST that would allow the user to create mappings internally within SYMBOLIST. The idea being that since the data is already in hierarchical format in the application data structure, it makes sense to be able to interpret the data internally, and then stream OSC containing the pre-interpreted data rather than the raw graphic information which required complex parsing of the graphic hierarchies to rebuild the symbolic hierarchies that were visible to the eye based on traditional common practice notation. Further, I realized that what SYMBOLIST *really* needed was a system for *bi-directional mapping*, where graphic data is interpreted as symbolic data with semantic meaning, and inversely, that the user should also be able to send symbolic semantic data to SYMBOLIST, which could then translate the semantic data to a graphic representation. In 2018 I began working to implement this idea into the JUCE version, but soon needed to switch tracks to focus on a different notation issue for a project at the Innovative Hochschule, developing a platform for realtime networked score display.

In 2019,

1.1 Clefs and Bidirectional Mapping

telling story of how we got where we are, conceptual steps
idea of Clefs
problem: a lot of mapping is still needed

2. DRAWSOCKET

At the end of 2018, and first half of 2019 we developed DRAWSOCKET
mention that in addition to the front end that the drawsocket message format [7] [8]

3. SYMBOLIST JS

using DRAWSOCKET as front end handler
a general overview about need for classes (actually this comes later when we get into the the discussion of format i.e. the way the data is represented.
js and DRAWSOCKET makes that easy(-ier)
Figure 1 shows a screenshot from the Electron version.

4. IMPLEMENTATIONS

Electron [10] version (node + chrome w/ special electron IPC)
Max version (node + jweb + Max patch for IPC)

4.1 Application Structure

SYMBOLIST is organized as a server-client model, comprising of:

1. The *editor*, a browser-based user interface client which displays the graphic representation of the data, and allows the user to edit and create new data objects through graphic interaction. The *ui_controller* runs in the browser (see Figure 2), and handles interaction via a library of definition scripts which specify mappings to and from data and graphics formats, as well as other tools and interactions.
2. The *symbolist server* runs in node.js (either in the context of Electron, or in Max's node.script object) and is comprised of:

[7] http://paperjs.org/
[8] https://juce.com/
[9] https://forum.juce.com/t/complex-svg-files-fail-to-load-properly/26917/16
[10] https://www.electronjs.org/

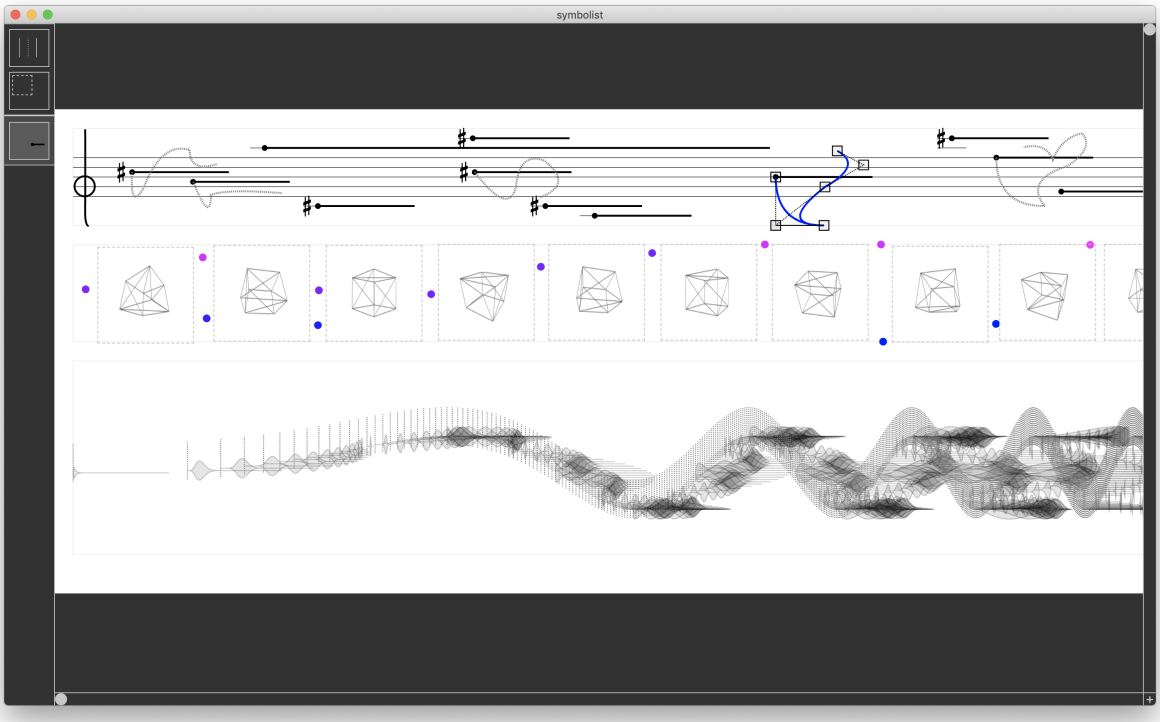

Figure 1. SYMBOLIST screenshot, showing some different types of staves, and editing capabilities.

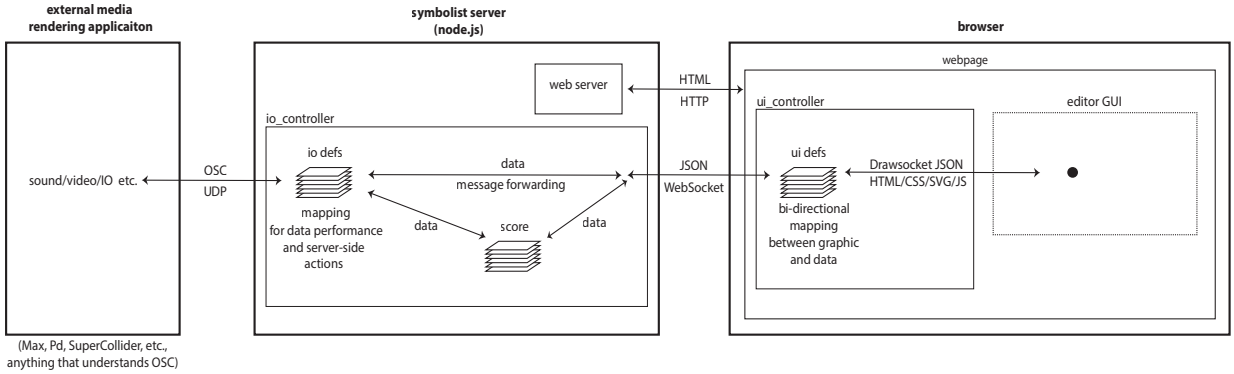

Figure 2. SYMBOLIST architecture.

- A HTTP web-server which serves the editor webpage, and manages messages between the *ui_controller* and the *io_controller* via WebSocket connection, as well as handling operating system commands like reading and writing files.
- The *io_controller* script runs inside the *symbolist server* and handles input and output from external sources via OSC over a UDP socket which can be used for a wide range of actions. The *io_controller* additionally maintains the central *score* database in its semantic data format (described below).

4.1.1 *Interprocess messaging syntax*

SYMBOLIST uses a *key/val* message syntax for OSC and JSON interprocess communication developed in DRAWSOCKET (discussed further in sections 17 and 16), where a *key* address is used a keyword to signal which routine should interpret the message, and the *val* address contains an object (or array of objects) to be processes.

5. GRAPHICAL AUTHORING AND INTERACTION

The graphic user interface of SYMBOLIST (Figure 1) is designed around the idea of symbol objects and containers. Graphic objects, or *symbols*, are placed in *container* references which define a framing used to interpret the meaning of the *symbol*.

In order to maintain an open and un-opinionated approach to authoring tools, SYMBOLIST tries not to specify how containers and symbols should look, act, or respond when

you interact with them within the application. Rather, the interaction and meanings of the symbols are defined in a library of custom object *definitions* which create these meanings through mapping semantic data to and from the graphic visualization. Definitions can be shared between users and provide a mechanism to setup tailored composition environments for different authoring situations.

5.1 Interface Components

Since the application is programmed using web-brower technologies (JS/HTML/CSS/SVG) there are many ways to customize the layout, using CSS, or defining new object types. The default SYMBOLIST graphic editor (Figure 1) provides the basic mechanisms for user interaction, to create scores through the creation of hierarchical object relationships. The main graphic components are:

- *Document view*: the top level view of the application window, containing the document, and side bar. Sliders are provided to offset the view of the document, as well as basic zoom functionality.
- *Palette*: a set of buttons on in the side bar of the program which display icons of the *symbols* that have been defined for the current selected *container*.
- *Tools*: a set of buttons that open tools for custom computer-assisted generation of new symbols, and applying transformations to existing elements (e.g. alignment of multiple objects, or setting distributing objects, etc.)
- *Inspector*: a contextual menu for editing the semantic data of an object, which is then mapped to the graphic representation.
- *Menu bar*: (Electron version only) the menu bar at the top of the screen or window, which provides access to various application functions.

5.2 Modes

In the process of working in SYMBOLIST, the user will shift between different modes, each of which have user-definable behaviors in the definitions library. The current modes are:

- *Palette*: clicking on an icon in the palette sidebar enables the user interaction assigned to the clicked symbol type.
- *Creation*: holding down the CMD key (Mac) enters *creation mode*, telling SYMBOLIST to create a new *symbol* of the type selected in the palette, similarly this action may enable different types of user interaction, for example snapping the symbol to specified pixels, etc.
- *Selection*: clicking on a pre-existing *symbol* in the document will *select* the object, which notifies a callback in the definition script for possible interaction.
- *Edit*: if a user has selected an object and then types the letter [e], SYMBOLIST will attempt to enter *edit mode* if there is one defined in the symbol definition script. This is useful for example in the case of a bezier curve; entering *edit mode* could make visible the handles for the curve for editing.
- *Inspector*: if a user has selected an object and then types the letter [i], an inspector window appears showing the *semantic data* corresponding to the symbol's graphical information (datatypes are discussed further in section 6 below).

5.3 User Experience

On entering the application, the editor loads a score or configuration file from the default load folder, which sets the top-level page setup and palette options. A typical sequence of creating a score might be as follows:

1. The user opens a workspace, with one or more default *container symbols* displayed on the screen, for example an empty rectangle which is like a piece of paper.
2. Selecting the "paper" container rectangle, the user then selects the container as the new *context* by pressing the [s] key (or from the application menu).
3. Once setting the context, the palette toolbar is populated with icons of symbols that are defined with the selected container context type.
4. Clicking on one of the palette toolbar symbol icons, puts the interface into *palette mode*, where the mouse interaction is now designed for use with this specific symbol type.
5. Holding the Mac CMD button enters *creation mode* and by default creates a preview of the symbol how it will appear when you click, and some text is displayed near the mouse that shows the semantic data associated with the graphic representation.
6. After clicking the symbol is placed in the container.
7. Depending on the symbol type, you may be able to drag the symbol to a new place in the container, and the associated data is updated as a result.
8. Selecting the symbol and hitting the [i] button, brings up the inspector window, where you can edit the data and see the graphics updated in response.
9. Selecting and pressing the [e] button enters *edit mode* which is a modal context where different user interaction could change the values of the symbol in different ways. For example in edit mode you might be able to rotate an object in a certain way, or be able to visualize different connections to the graphic representation to other elements of the score which are not usually highlighted in the score view.

6. DATA REPRESENTATION

At the heart of SYMBOLIST are two parallel forms of information expression: *semantic data* and *graphic representation* (Figure 3).

Semantic data specifies the various attributes of information about a symbolic object, in terms of the object's meaning to the author. For example, the meaningful attributes of a *note* object might be information about pitch and duration, or a *point* object might contain x, y, and z values corresponding to the point's location in 3D space. In SYMBOLIST *semantic data* is thought of as the main holder of information in the system, which through grouping and hierarchical arrangement can be used to represent scores or other types of data structures.

The *graphic representation* of the information is a visual expression of the semantic data, which is open in nature. The aim of SYMBOLIST is to provide an agnostic framework for developing visual, symbolic, or other unknown representations of semantic data for use in multimedia composition practice; and so, one of the main functions of the new version of the software is to facilitate the creation of mapping relationships between different representations of the data.

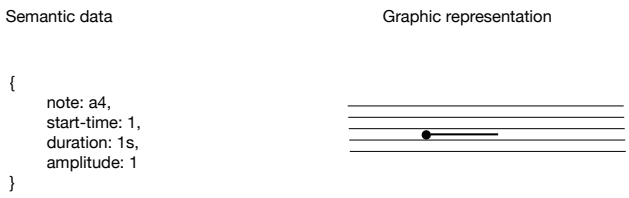

Figure 3. *data* vs *graphic* representation of the same information.

7. SYMBOLS

In SYMBOLIST terminology, a *symbol* is an instance of a symbolic representation of data that connects the semantic, graphic, and possibly other media types of expression together as a multifaceted unit. Each *symbol* is defined as a *class* of object, which specifies the symbol's data structure and UI interaction, and data mapping to different representational contexts.

8. SEMANTIC DATA FORMAT

Within the SYMBOLIST application, semantic data is stored as javascript objects, and read/written in JSON format [11], which is transcoded to and from OSC for inter-application communication.

The main attributes used in SYMBOLIST semantic data objects are:

- *id*: a unique identifier name (required).
- *class*: a reference to the definition of the object type in the user-definition library (required).
- *contents*: an array of child objects that a parent container object might hold (required for container symbols).

[11] https://www.json.org/json-en.html

In addition to the required *id* and *class* attributes, symbol objects may include any number of other *attributes* [12] of the symbol (*pitch*, *amplitude*, etc.). For example a simple semantic object written in JSON might look like:

```
{
   "id" : "foo",
   "class" : "legs",
   "action" : "jump",
   "start_time" : 0.1
}
```

Here we see an object with the *id* "foo," which is of *class* type "legs", that has an attribute *action* associated with it and a start time.

8.1 Containers

Symbols may also contain other symbols. Container symbols function to frame their contents, giving reference and context, like a plot graph frame, which provides a perspective and scaling for interpreting the set of data points displayed in the graph.

When a symbol contains other symbols, the child symbols are stored as an array in the object's *contents* field. For example an imaginary class "timeline", which holds two types of leg actions, we might write something like:

```
{
   "id" : "bar",
   "class" : "timeline",
   "duration" : 1,
   "contents" : [{
      "id" : "foo-1",
      "class" : "legs",
      "action" : "jump",
      "start_time" : 0.1
   },{
      "id" : "foo-2",
      "class" : "legs",
      "action" : "sit",
      "start_time" : 0.2
   }]
}
```

In most cases, a symbol's mapping definition will require querying its parent symbol for information, in order to plot its data relative to the container context, for example offsetting the screen coordinate position based on the parent object position.

9. SCORE FILE FORMAT

Using symbols and symbol containers, we can create tree structures which can be used to represent hierarchical grouping; to represent scores, or other types of data structures. At the root of the tree structure is a top-level symbol, which might (but not necessarily) define behavior of its children objects. Since the data elements are stored in js objects, it is easy to import/export SYMBOLIST scores as JSON files.

When the application loads, it reads a default initialization file, in the form of a SYMBOLIST score. The current default initialization config file looks like this:

[12] The term *attribute* is used here interchangeably with properties, parameters, aspects, etc.

```
{
   "about" : "some metatdata",
   "id" : "Score",
   "class" : "RootSymbol",
   "tools" : [],
   "palette" : ["SubdivisionTool", "BasicSymbolGL"],
   "contents": {
      "id" : "trio",
      "class" : "SystemContainer",
      "x": 200,
      "y": 100,
      "duration": 20,
      "time": 0,
      "contents" : [{
         "id" : "oboe",
         "class" : "FiveLineStave",
         "height" : 100,
         "lineSpacing" : 10,
         "duration": 20,
         "time": 0,
         "contents" : []
      },
      {
         "id" : "bassoon",
         "class" : "PartStave",
         "height" : 100,
         "time": 0,
         "duration": 20,
         "contents" : []
      },
      {
         "id" : "synth",
         "class" : "PartStave",
         "height" : 200,
         "time": 0,
         "duration": 20,
         "contents" : []
      }]
   }
}
```

The initialization file is literally a score object file, providing the default context for a given authoring situation. In this example, we can see there is a "RootSymbol", which contains a "SystemContainer", which in turn contains two "PartStave" symbols and one "FiveLineStave" symbol (which are all actually containers as well, initialized as empty arrays). The *palette* and *tools* attributes tell the application to provide access to certain tools, and child symbols in the GUI.

10. GRAPHIC DISPLAY FORMAT

The graphic representation of the data in SYMBOLIST uses SVG format, and and follows the same hierarchical structure of the data as found in the semantic data score object. Since the new version of SYMBOLIST uses a browser as a frontend, we are able to take advantage of the many standard tools and web functionalities provided by browsers for display, interaction and data management.

Like the JSON score initialization file above, the main application window is setup using HTML/CSS, and utilizing DRAWSOCKET as a convenience wrapper to provide shorthand methods to create and manipulating browser window elements.

10.1 SVG

The SYMBOLIST format for an SVG *symbol* is a one top group (<g>) elements, with two sub-groups for *display* and *contents*, using HTML/CSS class names. The most simple SVG symbol would be:

```
<g  id="foo" class="SymbolClassName symbol">
   <g class="SymbolClassName display"></g>
   <g class="SymbolClassName contents"></g>
</g>
```

Just like the semantic data objects, graphics objects have required *id* and *class* parameters, with an optional *contents* element.

Each symbol grouping element is tagged using class names, following the symbol's unique class name (in this example "SymbolClassName"). Note that the order is important: *the symbol class type must be first*. The *symbol* tag marks the top-level grouping object of the symbol, the *display* element is a group that holds all of this symbol's visual display information, and the *contents* is an group object for holding any potential child elements. Note that for simplicity, all SYMBOLIST graphic elements include the *contents* element as a placeholder.

10.2 HTML

Symbols and containers could also potentially be HTML elements instead of SVG. In the case of HTML you would use tags instead of SVG <g>: html:

```
<div class="SymbolClassName symbol">
   <div class="SymbolClassName display"></div>
   <div class="SymbolClassName contents"></div>
</div>
```

10.3 dataset-elements

Since SYMBOLIST is constantly mapping to and from semantic data and its graphic representation, we are using the HTML *dataset* feature [13] to store the semantic data inside the top-level *symbol* element (<g> or). The HTML dataset attributes use the prefix "*data-*". [14]

For example, mapping our imaginary "legs" actions above, the corresponding SVG objects would be (skipping the actual display drawing for now):

```
<g id="bar" class="Timeline symbol" data-duration="1">
 <g class="Timeline display"></g>
 <g class="Timeline contents">
   <g id="foo-1" class="Legs symbol" data-action="jump" data-
     start_time="0.1">
    <g class="Legs display"></g>
    <g class="Legs contents"></g>
   </g>
   <g id="foo-2" class="Legs symbol" data-action="sit" data-
     start_time="0.2">
    <g class="Legs display"></g>
    <g class="Legs contents"></g>
   </g>
  </g>
</g>
```

11. PERFORMING DATA

Just as the *graphic representation* can be seen as a visual expression of *semantic data*, the same semantic data can also used as control data in connection with other media forms. For example, a *note* object's pitch, onset, and duration information could be used to trigger a note on a synthesizer, or a sequence of Labanotation [9] could be used to guide the movement of robotic motors, create haptic feedback for live performance [10], and so on.

[13] https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/dataset

[14] Note that according to the HTML dataset specifications, all names will be converted to lowercase, this can create issues in some cases, so best practice is to use all lowercase for attribute names.

SYMBOLIST provides several different options for sorting and looking up data (see Figure 6), which can serve as a structure for the performance of a "score", or other data formats. Typically, some representation of time is used to indicate an object's moment of action, but in SYMBOLIST the exact nature of the temporal organization is up to to the author (see section **??** for further discussion of performing data).

In addition to the *semantic* and *graphic* contexts of data representation, we can think of the *performance* of the data as a third data context context.

12. MAPPING

Between each of these representation contexts there is a layer of mapping, with the *semantic data* serving as the primary representation type.

Semantic data to graphic representation mapping (Figure 4) is used for the creation of graphic symbols from a stream of input, for example from generative processes, textural authoring, or computer assisted composition systems [5, 11, 12, 13, 14].

Graphic representation to semantic data mapping (Figure 5) is used in order to create or edit data based on graphic information. This is the typical "graphical user interface" situation, where the data is accessible through its visual representation.

Semantic data to performance media mapping (Figure 6) is the use of the data as a sequence of events that can be played in time (or used to control other processes not necessarily in time).

Note that in SYMBOLIST mapping between *performance media* and *graphic representation* is achieved through first mapping to semantic data. See section 13 for further discussion.

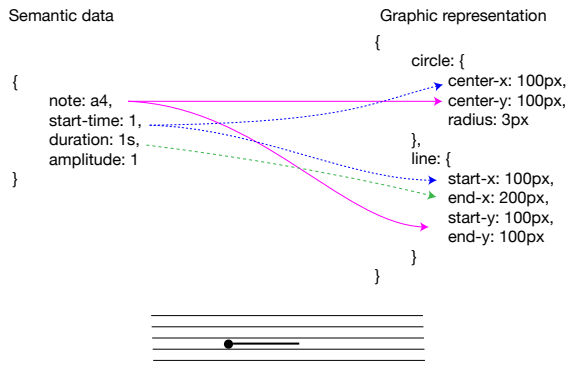

Figure 4. *semantic data* mapped to create a *graphic* representation from input data.

13. SYMBOL DEFINITIONS

Symbols are defined as Javascript classes which are stored and recalled when symbol actions are performed. For each user interaction, the ui- and io-controllers looks up the symbols involved in the interaction, and uses their definitions to enact the symbol's reaction. Definitions define the mapping relationships between the symbol's *semantic*, *graphic*, and output *performance* data. Through creating definitions, users are able to form libraries of symbols that can be used together to fit the tools and representation structure needed to address a given use-case scenario.

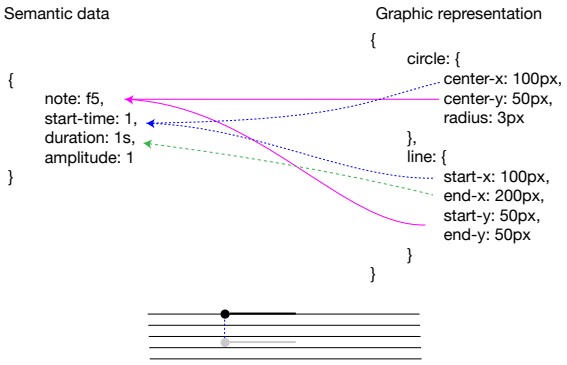

Figure 5. If edited graphically, the updated graphic data is then mapped back to *semantic data* representation.

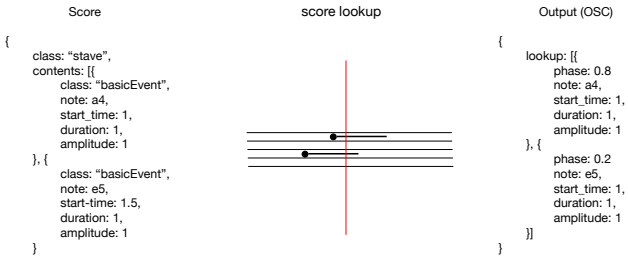

Figure 6. Using the lookup method defined by the symbol class, the *semantic data* can be used to perform external instruments via Open Sound Control.

To allow for maximum flexibility of interaction and the creation of context-specific composition environments, each symbol manages its own mouse interaction, triggered by the user's selection in the palette toolbar. In order to streamline the process of writing new symbol definitions there is a template base class that handles most common interaction situations, which can be redefined and overwritten by sub-classes. Eventually it is planned to provide a UI in the editor for defining a mapping definition graphically, but this is not yet implemented.

There are two types of definition scripts:

- *ui-definitions* run in the *ui_controller* and perform user interactions based on the different interaction modes described above, and apply bidirectional mapping between semantic data representation and graphic representation in the browser.
- *io-definitions* run in the *io_controller* and are used to assist in the lookup and *performance* mappings of the semantic data to media like sound synthesis, video, etc., or to perform server-side score manipulations.

In each controller context there are certain methods and variables that need to be defined in order for the class to function properly in the SYMBOLIST ecosystem.

IO and UI data access scope

Since the *ui* and *io* controllers are run in separate processes [15] (Figure 2), there is no direct access to data stored in the other location. In other words, the *io_controller* does not have direct access to the symbol's drawing information, and the *ui_controller* does not have direct access to the score or UDP port for sending OSC message.

Loosely following the MVC pattern, [16] the concerns of drawing are kept within the browser-side *ui_controller*, with all messages between the *ui* and *io* processes are in DRAWSOCKET JSON format, with all symbol data expressed in its *semantic* form. Meanwhile the *io_controller* manages the *score* and handles external OSC communication, relaying messages to the *ui_controller* as needed.

As discussed above, the graphic representation takes advantage of the HTML dataset feature, which provides a mechanism for storing the semantic data inside the graphic context (see section 10.3). Since ui-definitions are running in a web-browser, they are able to make use of the standard HTML-DOM JS methods for fast querying of elements (i.e. querySelector, getElementById, etc.) [17] to retrieve graphic as well as semantic data stored in the display hierarchy.

To provide similar data query access in the *io_controller*, the score data is stored in two JS objects: the *score*, which stores the semantic data in a hierarchical tree structure, and a object named *model*, which is a flat lookup table by unique id, with links to object references to the coordinated object in the score tree.

Since the ui- and io-controllers run in parallel they need to keep the other side updated in case of any alteration to the score data. Updates sent from the *io_controller* to the UI can be as simple as sending a new score which will trigger a redrawing of the graphics in the UI view. Updates from graphic user interaction require mapping from the graphic to semantic representation, which is handled in the browser. Any changes are sent back to the *io_controller* where the new semantic values are updated in the *model* (which automatically update the *score*, since both objects hold JS references to the same JS objects).

API Functions

In each controller context, there are a set of helper functions for use by symbol definitions stored in global objects called *ui_api* and *io_api*, which provide many essential operations.

14. UI DEFINITIONS

At the time of writing, the variables and methods defined in the symbol class referred to by the *ui_controller* are:

- *class*: the unique name of the symbol, used to store and lookup the symbol definition in the *ui_controller*.
- *palette*: in the case of container symbols, an array of class names of other symbols that can be used within this container, which are drawn in the palette toolbar when the user selects a symbol as a new context.
- *getPaletteIcon*: called when drawing the palette for a given container; returns an icon for display in the palette toolbar, using DRAWSOCKET format.
- *paletteSelected*: called when the user clicks on the palette icon for this symbol, used to trigger custom UI. When the symbol is selected in the palette, the definition should enable its mouse handers. For creating new symbols from mouse data (currently *cmd-click* is the convention).
- *getInfoDisplay*: called when creating the inspector window; returns drawing commands for the inspector contextual menu, for convenience *ui_api* provides a function called *makeDefaultInfoDisplay* which can be used in most cases.
- *fromData*: called when data is should be mapped to graphic representation. The definition is responsible for creating the graphic element, normally via DRAWSOCKET, but other approaches are also possible.
- *updateFromDataset*: called from the inspector when elements of the data should be updated. Usually this function will call *fromData* to redraw the graphic symbol, and should also send the updated data to the server.
- *selected*: called on selection and deselection, return true if selection is handled in the script, returning false (or no return) will trigger the default selection mechanics by the *ui_controller*.
- *drag*: called from *ui_controller* when the user drags selected symbols. For best results, the use the *ui_api translate* function to set the symbol's SVG translation matrix, but do not apply the translation until mouse up to avoid incremental state changes to score.
- *applyTransformToData*: called on mouse-up if selected objects have changed. Definition should then apply the transform matrix to the SVG attribute values. This is important because the attribute values not the translation matrix are used for mapping. The *ui_api* helper function *applyTransform* is provided for convenience.
- *currentContext*: called when the user enters or exits a container symbol (hitting the [s] key, [esc] to exit).
- *editMode*: called when entering and exiting edit mode.

14.1 Data and View Parameters

Probably the most important elements of the symbol definition is the bi-directional mapping between semantic and graphic forms.

Looking at Figures 4 and 5 we can see that in some cases the relationship between a semantic property and its graphic representation is not a one-to-one mapping. For instance in Figure 4 the *note* property needs to be mapped to a pixel position that is used for both the center point of a graphic

[15] and potentially separate devices.
[16] https://developer.mozilla.org/en-US/docs/Glossary/MVC
[17] https://developer.mozilla.org/en-US/docs/Web/API/Document

circle (note-head) as well as the starting point for a line (duration indication). In reverse, Figure 5 shows how when the user moves a symbol graphically, the new pixel positions need to be translated back into semantic data in order to update the score. This can get somewhat complex in cases of nonlinear mappings and hierarchical data structures.

In order to manage the mapping between semantic and display representation, the template base class uses an intermediate mapping stage called *view-parameters*. The idea is that the *view-parameter* stage contains the bare-minimum number of variables needed to draw the symbol.

For example, in Figure 4 the graphic representation requires a *y* position relative to the pitch, an *x* position relative to the start time, and a *width* value relative to the duration of the event (the amplitude is not displayed). After first mapping from the semantic attributes *note*, *start-time* and *duration* to view-parameters *x*, *y*, and *width*, the drawing method can then use the *x*, *y*, and *width* values to draw its two graphic objects from the reduced set of view-parameters values.

The ui template class uses two functions to define data-view mappings: *dataToViewParams* which receives the semantic data object and returns the view-parameter object, and *viewParamsToData* which performs the opposite mapping. Note that in many cases the *viewParamsToData* function needs only one aspect of the graphic to map back to semantic data. In the example shown in Figure 5, the mapping only really needs either the center point of the note-head or the start-x position of the line to determine the *start-time* parameter.

The template class also two additional data/view parameter translation methods to coordinate child objects with parent containers: *childDataToViewParams* and *childViewParamsToData*. For example, in Figures 4 a note-head circle is drawn from its *note* parameter, in coordination with a five-line staff. Like a plot graph, the *staff* is a container symbol which defines how we interpret the elements written on its lines. In this case, the symbol definition for the *staff* has a *childDataToViewParams* function is called by the action of the child symbol. The *childDataToViewParams* receives the child data object, as well as the graphic element of a particular staff (which was selected by the user pressing the [s] key). Given its placement on the screen and its own data parameters (number of lines, clef, etc.) the staff's *childDataToViewParams* function will return the view parameters for the child, mapped in relationship to the container object. Similarly, when the graphic object is moved, the child object will call the parent's *childViewParamsToData* function to assist with mapping back to semantic data format.

15. IO DEFINITIONS

At the time of writing, the *io_controller* has a much simpler function, which is to handle OSC messages from external applications, score data maintenance, file reading and writing, and responding to external score lookup queries for sequencing. Variables and methods currently used by the *io_controller* are:

- *class*: class name, corresponding to class name in UI Definition, used to store and lookup the symbol definition.
- *comparator*: a comparator function used in container symbols to sort child symbol. For example, if a given container uses a *time* value for sorting, when a new child node is added, the comparator function helps the container insert the child element at the correct location in the *contents* array. This assists to increase the efficiency of looking up events occurring at a given time (or other data sequencing parameter).
- *lookup*: called via OSC (using the DRAWSOCKET syntax *key* "lookup") to look up events at a given value specified by the container (e.g. typically *time*). The *lookup* function performs hit detection collecting all active elements at that query point, output back to the calling application via OSC (the in/out UDP port information is set in the SYMBOLIST configuration file). By default the output is an array of all active data objects at the lookup point, along with the relative phase position within each element, useful for controlling amplitude envelopes etc. (See Figure 6).
- *getFormattedLookup*: called via OSC to request a complete list of events for external sequencing, formatted in the symbol definition to apply to the external syntax requirements.

Note that the *lookup* and *getFormattedLookup* methods are able to view the whole OSC bundle that is sent in, and have access to the entire score data model, and so it is also possible to define multiple ways of looking up (and performing) the score data at the same time; or example multidimensional nearest neighbor lookup, or polytemporal sequencing, etc.

16. CREATING SYMBOLS FROM OSC INPUT

As an illustration of how data is processed through the SYMBOLIST architecture, we can follow the sequence of actions taken in the case of *semantic to graphic* mapping; for example when algorithmically generating score data, using an outside process to create the data and sending it to SYMBOLIST via OSC, using the DRAWSOCKET key/val syntax.

Data input via OSC

The *io_controller* has a small collection of built-in processes that can be called via OSC, the most important of which is the function to add new data elements to the score and graphic display, accessible using the *data* keyword.

For example, here is an OSC bundle using the *data* key:

```
{
   /key : "data",
   /val : {
      /class : "FiveLineStaveEvent",
      /id : "foo"
      /container : "oboe",
      /time : 0.13622,
      /ratio : "7/4",
      /duration : 0.1,
```

```
    /amp : 1
  }
}
```

The "data" keyword message has the following required and optional attributes:

- *class*: the class name of the object to create (required) .
- *container*: the *id* of the container symbol class to put the object in (required).
- *id*: a unique id to use for the data object (optional); if not specified a unique string will be generated .
- Other required or optional parameters will depend on the symbol definition.

Upon receiving an OSC message with the *key* "data", the object payload stored by *val* is added to the model, and then relayed to the *ui_controller*.

Data to View Mapping in the ui_controller

Received by the *ui_controller*, the semantic data then is mapped to graphic data, by looking up the symbol's *class* definition and calling the ui-definition's *fromData* method, which maps from the data representation to the graphic drawing commands.

As discussed above (in Section 14.1), when using the symbol template base-class, the *fromData* method will usually call the symbol's internal *dataToViewParams* which performs the mapping from semantic to a minimal set of graphic values which are then used to draw the graphics, by sending drawing commands to DRAWSOCKET accessed through the *ui_api*, including the HTML dataset storage, as described above (in Section 10).

A typical drawing command would look something like:

```
ui_api.drawsocketInput({
    key: "svg",
    val: {
        class: `${className} symbol`,
        id: uniqueID,
        parent: container.id,
        ...newView,
        ...ui_api.dataToHTML(dataObj)
    }
})
```

Here, we use the JS spread operator "..." to merge the *newView* variable, holding DRAWSOCKET format SVG data, and the HTML dataset information, encoded via the *ui_api dataToHTML* helper function into the *val* object with the associated "svg" DRAWSOCKET keyword. The object is then sent to DRAWSOCKET via the *ui_api drawsocketInput* helper function to be added to the browser screen.

17. CUSTOM USER METHODS

Users may also create their own additional methods in either ui or io-definitions and call them from outside processes over OSC, or from other symbol definitions, using the "call" keyword. [18]

[18] SYMBOLIST will pass the same call request to both definitions, so if both have a function of the same name they will both be called.

As elsewhere, SYMBOLIST uses DRAWSOCKET syntax, and requires two parameters in the *val* object need to lookup and execute the method:

- *class*: name of the class to lookup.
- *method* name of class method to call.

All of the parameters in the *val* object will be passed to the function as an argument.

User class methods can be used to apply operations to the score or ui, for example transposing all pitches on the "Staff" named "oboe" might look like this:

```
{
  /key : "call",
  /val : {
    /class : "Staff",
    /method : "transpose",
    /id : "oboe"
    /steps : 12
  }
}
```

On receiving this OSC bundle, the *io_controller* will lookup the class "Staff" and attempt to call its method "transpose", passing the entire *val* object to the symbol method as an argument. The user-defined "transpose" function might then do something like lookup the "oboe" staff in the model, and then iterate all of its contents, offsetting the "note" values by the number of steps specified in the method arguments.

18. CONCLUSIONS

With the new symbol class definition system in place, initial experiments indicate that this new experimental SYMBOLIST implementation should be able to handle a very wide variety of score and symbol structures, and provide the mechanisms for users to compose bi-directional mappings between semantic and graphic representation. The system seems to have a potential for many applications in digital media compositional practice, and may someday evolve into a fully functional authoring environment for computer performable symbolic notation.

In order to further evaluate the robustness of the system, the next step will be to go through the process of developing complete definition libraries for working with different types of notation systems. As a test case, currently one of the PhD students at the HfMT Hamburg has been working on a set of definitions for common practice notation, which is planned for presentation at the 2023 TENOR conference, along with other experimental approaches.

One challenge that may need to be addressed is the ease or difficulty there is in creating new symbol definitions. At the moment the system is based in Javascript, which means that the user must program the definitions with textural code. However, as a graphic oriented authoring environment, it would be convenient if there were some way to create new symbol definitions graphically, or at least within the SYMBOLIST graphical environment. In the process of creating test definition libraries, further research is planned to develop a GUI for symbol definitions, and if possibly streamline the process of bidirectional mapping.

For example, most mathematical operations have an inverse operation, and so perhaps there could be a GUI interface that provides tools to define both mapping directions simultaneously.

The Electron framework is currently working well for cross platform app development, however there have been some issues that came up after Electron version 12 which introduced new security measures, including context isolation [19] , and increase limitation of the using the require function to import user libraries. In order to function in a web-safe way, SYMBOLIST is currently using Webpack [20] to bundle the JS classes into a single file that is loaded on startup. Previously, users were able to dynamically load symbol definitions at run time, which seems like a more natural application working model. However, since SYMBOLIST is now browser-based and using the same system as DRAWSOCKET for dynamic graphic rendering, there is also the possibility of networked use of SYMBOLIST, for example, in connection with a graphic multitouch interface (iPad etc.), it would be possible to interact with SYMBOLIST over a network, and therefore possibly the web-security measures are important. More testing is needed to determine which features are the most important.

For playback/sequencing of SYMBOLIST scores, currently users can send either the *lookup* or *lookupFormatted* messages to the *io_controller*, which will then respond with data that can be used to perform the score in another software like Max, Pd, SuperCollider, etc. The lookup is currently implemented in Javascript, which is not the fastest or most temporally precise method of playing back the score. As a starting point, a Max external called *o.lookup~*, which accepts a list of x and y coordinate points and reads through the sequence of points via a sample-rate phase input. This system works quite will for single data sequences (i.e. value of y at point x), however for more robust playback, it might be worthwhile to develop a C/C++ based database lookup system, which could provide optimized getter methods for data playback. For example, this might take the form of a Max external that can read a SYMBOLIST score and provide optimized access for playback (possibly updated in realtime via connection to the SYMBOLIST server). Or even further, it could be imagined that a score could be exported to playback in a DAW like Ableton Live. It could also be possible to attach links to other OSC sequencing applications like IRCAM's Antescofo expression language[15].

Other development directions that may be interesting to pursue would be to integrate other frameworks into the application. Some first steps for 3D graphics have begun with the introduction of the three.js [21] library, visible in the rotated cubes in Figure 1, however more work is needed to provide tools for manipulating 3D graphics. In the area of notation for spatial movement, there are plans to continue development of trajectories (visible in the curves attached to note events in Figure 1, and to connect SYMBOLIST with the ICST's Spatialization Symbolic Music Notation (SSMN)[16].

In the audio domain, it could be interesting to develop tools for development of signal processing graphs that could be interpreted and performed in other applications, for example generating Faust [22] or Gen~ DSP code, SuperCollider instruments etc.

There are very many possibilities for the future development of SYMBOLIST, and so far it seems that the framework is providing a solid ground for the creation of new authoring environments. In a way, SYMBOLIST is a meta-environment, an application that aims to ease the process of creating new authoring environments. Like the creation of a new instrument, the challenge then is to work through the difficulties of creating the instrument, so that the instrument can be learned to play, and then, finally, the instrument can be used for the creation of new kinds of art.

[19] https://www.electronjs.org/docs/latest/tutorial/context-isolation

[20] https://webpack.js.org/

[21] https://threejs.org/

[22] https://faust.grame.fr/

Acknowledgments

A special thanks to James Tsz-Him Cheung for his work on the common practice notation definitions, which is greatly helping push SYMBOLIST forward, and especially thank you to Prof. Dr. Georg Hajdu for his collaboration and ongoing support for this work.

19. REFERENCES

[1] M. Wright, "Open Sound Control: an enabling technology for musical networking," *Organised Sound*, vol. 10, no. 3, pp. 193–200, 2005.

[2] J. MacCallum, R. Gottfried, I. Rostovtsev, J. Bresson, and A. Freed, "Dynamic Message-Oriented Middleware with Open Sound Control and Odot," in *Proceedings of the International Computer Music Conference (ICMC'15)*, Denton, TX, USA, 2015.

[3] R. Gottfried, "Studies on the Compositional Use of Space," IRCAM, Paris, France, Tech. Rep., 2013.

[4] ——, "SVG to OSC Transcoding: Towards a Platform for Notational Praxis and Electronic Performance," in *Proceedings of the International Conference on Technologies for Notation and Representation (TENOR'15)*, Paris, France, 2015.

[5] J. Bresson, C. Agon, and G. Assayag, "OpenMusic: Visual Programming Environment for Music Composition, Analysis and Research," in *Proceedings of the ACM international conference on Multimedia – Open-Source Software Competition*, Scottsdale, AZ, USA, 2011, pp. 743–746.

[6] R. Gottfried and J. Bresson, "Symbolist: An open authoring environment for end-user symbolic notation," in *International Conference on Technologies for Music Notation and Representation (TENOR'18)*, 2018.

[7] R. Gottfried and G. Hajdu, "Drawsocket: A browser based system for networked score display," in *Proceedings of the International Conference on Technologies for Music Notation and Representation–TENOR 2019*.

Melbourne, Australia: Monash University, 2019, pp. 15–25.

[8] G. Hajdu, "Quintet. net: An environment for composing and performing music on the internet," *Leonardo*, vol. 38, no. 1, pp. 23–30, 2005.

[9] A. H. Guest, *Labanotation: The System of Analyzing and Recording Movement*. Routledge, 2005.

[10] T. J. West, A. Bachmayer, S. Bhagwati, J. Berzowska, and M. M. Wanderley, "The design of the body: suit: score, a full-body vibrotactile musical score," in *International Conference on Human-Computer Interaction*. Springer, 2019, pp. 70–89.

[11] N. Didkovsky and G. Hajdu, "MaxScore: Music Notation in Max/MSP," in *Proceedings of the International Computer Music Conference (ICMC'08)*, Belfast, Northern Ireland / UK, 2008.

[12] A. Agostini and D. Ghisi, "A max library for musical notation and computer-aided composition," *Computer Music Journal*, vol. 39, no. 2, pp. 11–27, 2015.

[13] T. Baca, J. W. Oberholtzer, J. Treviño, and V. Adán, "Abjad: An open-source software system for formalized score control," in *2015. Proceedings of the First International Conference on Technologies for Music Notation and Representation-TENOR2015. Paris: Institut de Recherche en Musicologie*, 2015, pp. 162–169.

[14] G. Burloiu, A. Cont, and C. Poncelet, "A visual framework for dynamic mixed music notation," *Journal of New Music Research*, vol. 46, no. 1, pp. 54–73, 2017.

[15] J.-L. Giavitto, J.-M. Echeveste, A. Cont, and P. Cuvillier, "Time, timelines and temporal scopes in the antescofo dsl v1. 0," in *International Computer Music Conference (ICMC)*, 2017.

[16] E. Ellberger, G. T. Perez, J. Schuett, G. Zoia, and L. Cavaliero, *Spatialization Symbolic Music Notation at ICST*. Ann Arbor, MI: Michigan Publishing, University of Michigan Library, 2014.