SYMBOLIST RE-IMAGINED: BIDIRECTIONAL GRAPHIC-SEMANTIC MAPPING FOR MEDIA NOTATION AUTHORING AND PERFORMANCE

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

ABSTRACT

SYMBOLIST is an in-development application for experimental notation, which aims to provide an un-opinionated authoring environment for the design and performance of symbolic notation. By following an information visualization rather than prescribed musical orientation, the application is thought of as an open play-space, with tools for experimentation and thinking visually about relationships between representation and interpretation in media performance. In the paper we begin with an overview of the project's background, iterations and relationship to the DRAWSOCKET project, and introduce a redesign of the system, centered on a new framework for custom symbol definitions for bidirectional mapping and user interaction. In conclusion we discuss future development directions and evaluation of the project.

1. BACKGROUND

The origins of the SYMBOLIST project can be traced back to 2011, through the development of a composition practice in Adobe Illustrator using a plugin called Scriptographer, [1] which allowed users to create new drawing tools in Javascript which could then be used in Illustrator as an interactive brush. As a composer working with experimental instrumental techniques and spatial notation, Scriptographer was perfect for my composition needs at the time, since you could design a notation for a given technique or musical expression and then code it as an interactive graphic function, which could then be manipulated graphically in Illustrator. With the access to the mouse movement, interactions could be used to compose different elements of the notation, much in the same way that mouse interaction is used in programs like Processing. [2]

For example, a note with a spatially indicated duration typically is written as a note-head of some shape with a line extending out from it to show its duration. Using Scriptographer, you could create an interaction for composing note-duration symbols, where clicking down on the Illustrator canvas places the note-head, and then using the mouse drag, determine the end of the duration line, ending at the location of the mouse up event.

Further, you could also group elements to structure hierarchies of objects which would then appear in Illustrator as grouped objects, visible as nested folders in the layers menu.

Around the same time, I was studying at UC Berkeley's Center for New Music and Audio Technologies (CNMAT) developing approaches to instrument design using OpenSoundControl (OSC) [1] for data structuring. One day, while working with Scriptographer, I had saved a score in Illustrator as Scalable Vector Graphics (SVG) format [3] and accidentally opened the SVG file in a text editor. In the SVG file I noticed that all of the graphic objects were 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 might be able to translate the SVG information into OSC, then "perform" the OSC score in much the same way as you would a stream of OSC coming from a sensor-based instrument—in a similar spirit to Daphne Oram's "Oramics" [3] and Xenakis' UPIC system [4], but with a greater focus on symbolic interpretation.

Soon after, while composing for a high-resolution spatial audio rendering system, I found that I was lacking a way to compose spatial movements graphically, in a way that would connect with instrumental notation practice. After some experiments using Blender [4] to draw 3D curves which could then be parsed via Python and sent out over OSC, I was dissatisfied by the perceptual differences between common practice notation and the kinds of 3D representation I was able to create in Blender—both had their merits, but what was missing was a compositional frame that connected the two representation paradigms into a unified notation system. Due to time constraints, for this piece 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 practical, but this kind of automation approach has the limitation of forcing the composer to separate each data parameter into separate streams of data (e.g. x, y, z), whereas in a symbolic representation multiple attributes can be indicated in unified graphic representation. Following these experiences [5] I later returned

[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/

to the SVG-OSC transcoding idea, and developed a first working model which was presented at the 2015 TENOR conference [6].

In the meantime, the Scriptographer project was abandoned by its developers after Adobe drastically changed their plugin API in version CS6. The new Adobe API was different enough that it would require a significant amount of work, and even then the mouse interaction tools that were used in Scriptographer were no longer accessible, so the authors decided to stop development on the project and later went on to create Paper.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.

1.1 Symbolist JUCE

After preliminary tests using the SVG-OSC transcoding for score playback, it was becoming increasingly complicated to parse complex hierarchies of symbols in order to format them into OSC streams, so in 2017 I began work in collaboration with OpenMusic [7] developer Jean Bresson, through an 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.

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 [8].

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. There were some minor complications, due to JUCE's incomplete SVG support, [9] however, generally, we were able to create a working first prototype of the system.

1.2 Clefs and Bidirectional Mapping

A conceptual turning point in the project occurred towards the end of the residency as we were thinking about how to simplify the process of using the SVG data for controlling digital processes.

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 map the data to other processes, for instance synthesis parameters or coordinates for spatial rendering.

The process of interpretation requires that the parsing-mapping algorithm knows the context of the graphic objects. For example, that a circle is a note-head and not a rhythmic dot, and so on. Like the axes of a graphic plot of information, contextual musical symbols (meter, staff-lines, 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 from the idea of the *clef* as a graphic symbol that contextualizes the notation on a musical staff, I began work on developing a system for SYMBOLIST that would allow users to define interpretive symbols for hierarchical context parsing. Since the data is already in hierarchical format inside the application's data structure, it was logical that SYMBOLIST could interpret the graphics internally, and then stream OSC post parsing and mapping, rather than streaming the raw graphic information which required complex parsing of the graphic hierarchies to contextualize the symbol references, as in traditional common practice notation.

Following this line of thought it became clear that what SYMBOLIST really needed was a system for *bidirectional mapping*, where graphic data is interpreted as symbolic data with semantic meaning, and inversely, that the user should also be able to send data in its semantic representation to SYMBOLIST, where it would then be mapped to its graphic representation.

In 2018 I began implementing this idea into the JUCE version of SYMBOLIST, 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.

1.3 Drawsocket and Symbolist JS

At the end of 2018, and first half of 2019 we developed a drawsocket, a server/client framework for realtime dynamic notation using web browsers for graphic rendering [9, 10]. Based in node.js and standard web technologies of HTML, SVG, CSS, and Javascript, DRAWSOCKET is essentially an OSC wrapper [11] for web browsers, [10] providing a homogeneous message API for the creation and realtime manipulation of browser elements.

Returning to SYMBOLIST after the first DRAWSOCKET concerts, I began wondering if the framework of SYMBOLIST needed to be redesigned from the perspective of bidirectional mapping, since I realized that this was most likely going to be the most important part of the graphic-data relationship that the system is developing towards.

SVG is very well supported in modern browsers, and having just worked with node.js and browser technologies for DRAWSOCKET, it seemed that the flexibility of Javascript could be a convenient option for users to create custom symbol definitions (as in Scriptographer). I decided to see how fast it would be to implement a proof of concept using Electron.js, [11] a cross-platform desktop application development using a node.js server and Chrome as a front-end, used for applications like Skype, WhatsApp, Visual Studio Code, Slack, WordPress, and others.

[7] http://paperjs.org/

[8] https://juce.com/

[9] https://forum.juce.com/t/complex-svg-files-fail-to-load-properly/26917/16

[10] And by extension, provides access to other media via WebAudio, WebGL, etc.

[11] https://www.electronjs.org/

Using DRAWSOCKET as a frontend, and node.js for the backend I found that I was able to get up and running very quickly with Electron, and decided to continue SYMBOLIST development in this direction, bringing the experience gained from the JUCE version to the creation of a deeper structure for the creation of symbol interpretations that integrate into the graphic manipulation of symbolic data.

2. APPLICATION STRUCTURE

The new implementation of SYMBOLIST is organized as a server-client model (Figure 2), comprising of:

- The main SYMBOLIST application server, running in node.js, which serves the main display page and manages messages between the client and server via WebSocket [12] connection. Within the main server, a child process called the "io-controller" handles input and output from external sources via OSC over a UDP socket, and maintains the "*score*," a database of hierarchical score elements, stored in SYMBOLIST "*semantic representation*" format (Section 4).
- 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 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.

Since the system is now based on a web-server model, we are able to also use Max's *node.script* object to run the server from within Max as an alternative to the Electron desktop app.

3. GRAPHICAL AUTHORING AND INTERACTION

The SYMBOLIST graphic user interface (Figure 1) is designed around units of symbolic objects and their contextual containers. Graphic "*symbol*" objects are placed in "*container*" symbols, which function as a context frame that can be used to interpret the meaning of the symbol.

In order to maintain an open and un-opinionated approach, the SYMBOLIST framework does not specify how containers and symbols should look, act, or respond when you interact with them. Rather, the interaction and meanings of the symbols are defined in a library of custom object "*definitions*," which specify meaning through mapping semantic data to-and-from its graphic representation.

Symbol definitions provide a mechanism to design custom-tailored composition environments for particular authoring situations, stored as Javascript libraries, which can be shared between users, and used as templates. Leveraging web-browser technologies like JS, HTML, CSS, SVG, etc., there are many ways to customize the layout in SYMBOLIST, and potentially, the definition libraries could completely transform the editor layout to serve a particular use-case scenario.

[12] https://websockets.spec.whatwg.org

3.1 Interface Components

The main graphic components of the default SYMBOLIST graphic editor are:

- *Score view*: the top-level view of the application window, containing the main view and side bar. Sliders are provided to offset the view of the document, as well as basic zoom functionality.
- *Palette*: a set of buttons in a side toolbar displaying icons of symbols that have been defined for the current selected container context.
- *Tools*: a set of buttons that open high-level tools, that can be used for operations like algorithmic generation of new symbols, or applying transformations to existing elements (e.g. alignment of multiple objects, setting distributing objects, or other operations).
- *Inspector*: a contextual menu for editing the semantic data of a selected symbol, which on update is mapped to the graphic representation and sent to the server to update the main score database.

3.2 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. Clicking on the "paper" container rectangle *selects* it, and then the user sets it as the new *context* by pressing the [s] key.
3. After 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 Command(Mac)/Control(Win) key enters "*creation mode*," which by convention draws a temporary preview of the symbol (how it will appear when you click), and displays the corresponding semantic representation data as textual feedback.
6. After clicking, the symbol is placed in the container.
7. Clicking and dragging a symbol graphically modifies its semantic data in reference to the container context. In the case of common practice notation this would be how you would change the time and pitch information of a symbol. The interaction results depend on the "*selection mode*" specified in the symbol definition.

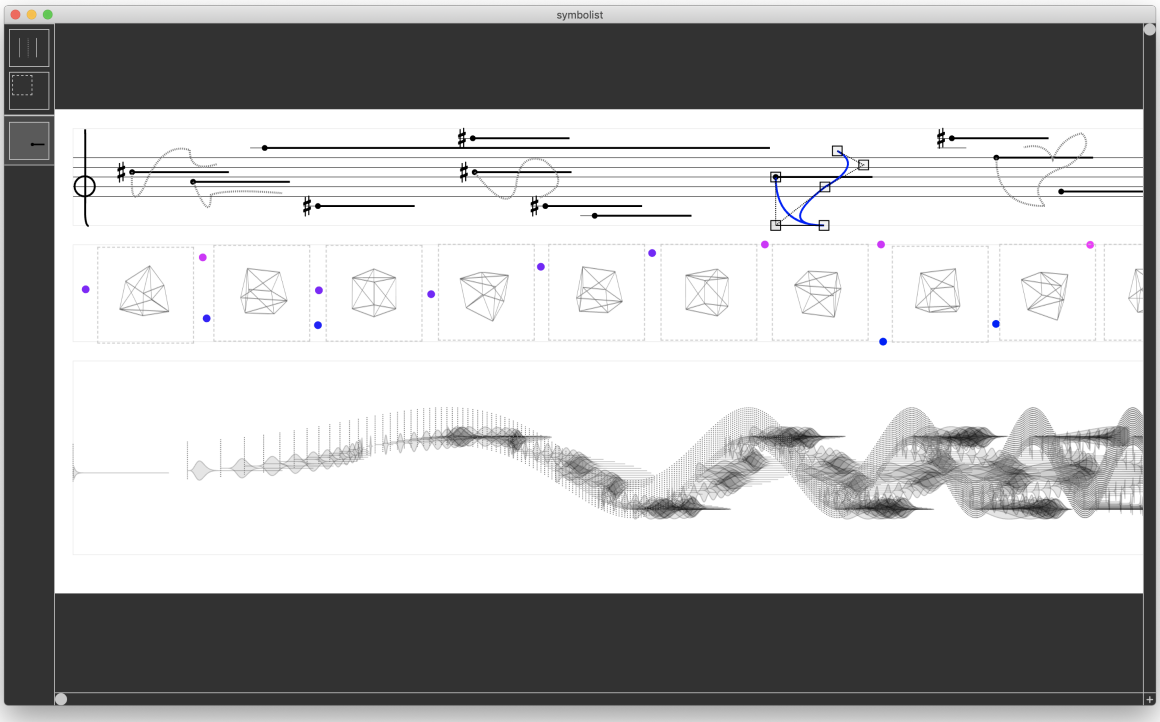

Figure 1. SYMBOLIST screenshot, showing some different types of staves, and editing capabilities.

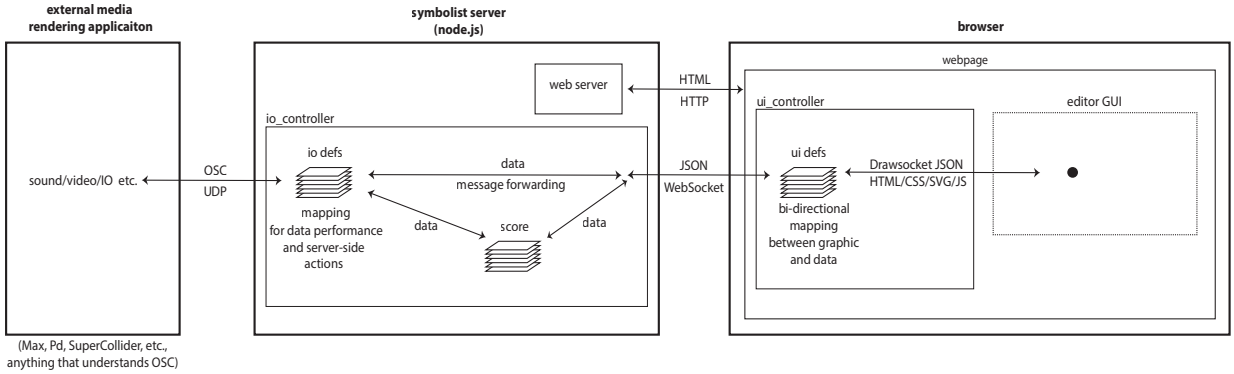

Figure 2. SYMBOLIST architecture.

8. User can also modify the semantic parameters as text by selecting the symbol and hitting the [i] key, which brings up the inspector window, where you can edit the data directly.
9. Pressing the [e] key enters "*edit mode*" for the selected symbol, useful for editing of internal attributes that are less relative to the container symbol. For example, you would enter edit mode to adjust bezier curve anchor points.

4. DATA REPRESENTATION

At the heart of SYMBOLIST are two parallel forms of information expression: *semantic* 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 the semantic representation is thought of as the main holder of information, which can be grouped into hierarchies to represent scores or other types of data structures.

The *graphic* representation is a symbolic visual expression of the semantic data, designed relative to the context defined by the author.

The aim of SYMBOLIST is to provide an agnostic environment for developing, and composing with, new symbolic representations of semantic data for use in multimedia composition practice; and so, the central design con-

sideration of this new implementation is to build a flexible framework for specifying a wide range of mapping relationships between semantic and graphic representations.

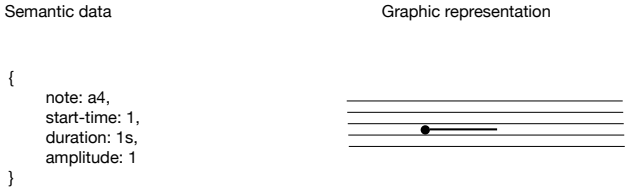

Figure 3. *Semantic* vs *graphic* representation of the same information. Note: Figures 3-6 use pseudocode for brevity (see Sections 6 and 7 for syntax details).

5. 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 [7, 12, 13, 14, 15].

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 8 for further discussion.

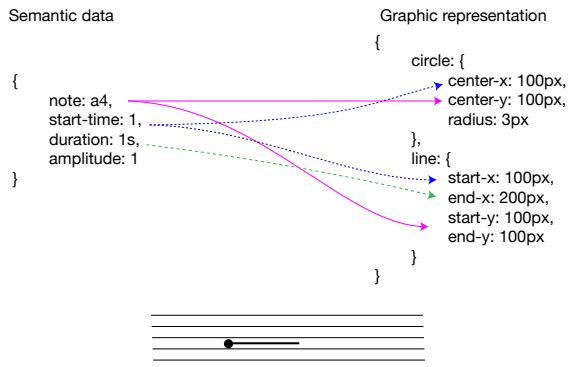

Figure 4. *Semantic* data mapped to create a *graphic* representation from input data.

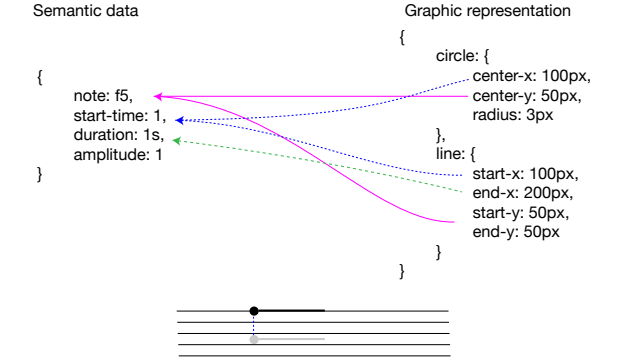

Figure 5. If edited graphically, the updated graphic data is then mapped back to *semantic data* representation.

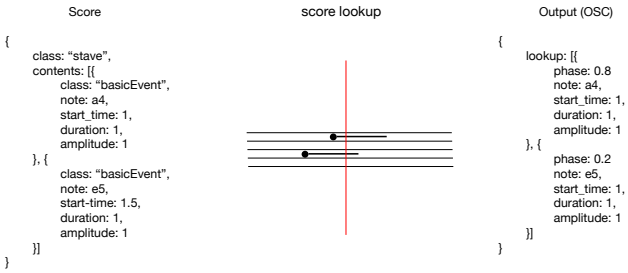

Figure 6. Using the lookup method defined by the symbol class, the *semantic data* can be used to perform external instruments via Open Sound Control.

6. SEMANTIC REPRESENTATION

Within the SYMBOLIST application semantic data is stored as Javascript objects and read/written in JSON [13] format (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).

Semantic data objects may include any number of other attributes [14] (*pitch*, *amplitude*, etc.). For example a simple semantic object 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* "legs," that has an attribute *action* associated with it and a start time.

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

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

6.1 Containers

Symbols may also contain other symbols. Container symbols function to frame their contents, providing 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. Similarly, when a semantic data object contains other objects, the children are stored as an array in the object's *contents* field. For example, for an imaginary class "timeline," which holds two types of leg actions, we might write:

```
{
  "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
  }]
}
```

6.2 Score Files

Since the data elements are stored as 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:

```
{
  "about" : "some metatdata",
  "id" : "Score",
  "class" : "RootSymbol",
  "contents": {
    "id" : "trio",
    "class" : "SystemContainer",
    "x": 200,
    "y": 100,
    "duration": 20,
    "time": 0,
    "contents" : [{
      "id" : "oboe",
      "class" : "FiveLineStave",
      "height" : 100,
      "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" : []
    }]
  }
}
```

In this example, we can see there is a "RootSymbol," which contains a "SystemContainer," which in turn contains two "PartStave" symbols and one "FiveLineStave" symbols.

7. BROWSER NOTATION

SYMBOLIST uses SVG to draw graphic symbol representation, utilizing DRAWSOCKET as a convenience wrapper to provide shorthand methods for the creation and manipulation of browser window elements.

7.1 SVG / HTML Format

The SYMBOLIST format for a *symbol* in its browser rendered notation, is a set of three group elements (<g> in SVG, or in HTML) marked by *class* tags, which follow the defined symbol class name.

For a symbol class type "SymbolClassName," the SVG template would be:

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

The "*symbol*" class marks the top-level symbol group containing the "*display*" and "*contents*" groups. The "*display*" group holds all of the symbol's display information and the "*contents*" group contains any potential child elements. For simplicity all graphic *symbol* elements include both the *display* and *contents* elements as placeholders.

7.2 Storing Semantic Data in Dataset Attributes

Since SYMBOLIST is constantly mapping back and forth between semantic data and its graphic representation, we are making use of the HTML *dataset* feature [15] to store the semantic metadata inside the top-level *symbol* element.

For example, using our imaginary "legs" actions above, we include the *action* and *start_time* parameters, written as dataset attributes by using the prefix "*data-*": [16]

```
<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>
```

8. 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 look up the symbols involved in the interaction by class name, and use their definition to execute the symbol's reaction.

Definitions specify the bidirectional mapping between *semantic* and *graphic* representations and responses to OSC "*lookup*" queries which can be used to perform the score.

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

[16] 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.

There are two types of definition scripts:

- *ui-definitions* run in the ui-controller and perform user interactions based on the different interaction modes, and applies bidirectional mapping between semantic and graphic representations.
- *io-definitions* run in the io-controller and are used to assist in the lookup and OSC *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. Users may also call custom io- and ui- methods from external applications via OSC (described further in Sections 11 and 12).

For convenience, there is a base class template that can be used to handle most common interaction situations, which may be overridden by sub-classes for custom handlers. A set of helper functions are provided in global objects called "*ui_api*" and "*io_api*" which can be used in definitions for many essential operations. Eventually it is planned to create a graphical tool in the editor to help define symbol definitions, but this is not yet implemented.

9. UI DEFINITIONS

At the time of writing, the variables and methods defined in the symbol class used in the the ui-controller when handing user actions are:

- *class*: the unique name of the symbol, used to store and lookup the symbol definition.
- *palette*: an array of class names of other symbols that can be used within a container symbol, which are drawn in the palette toolbar when the user selects the symbol as a new context.
- *getPaletteIcon*: called when drawing the palette icon, returns DRAWSOCKET drawing commands.
- *paletteSelected*: called when the user clicks on the symbol icon, used to trigger custom UI. When the symbol is selected in the palette, the definition should enable its mouse handers.
- *getInfoDisplay*: called when creating the inspector window; returns drawing commands for the inspector contextual menu.
- *fromData*: called to map data from semantic to graphic representation.
- *selected*: called on selection and deselection, for optional custom UI handling.
- *drag*: called when the user drags selected symbols; by default the *ui_api translate* function is used to set the symbol's SVG translation matrix to preview the new location.
- *applyTransformToData*: called on mouse-up if selected objects have changed, and applies the transform matrix to the SVG attribute values.
- *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.

9.1 Data and View Parameters

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 both for the center point of a graphic 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 to its semantic representation to update the score.

To manage the bidirectional mapping between semantic and display representation, the template base class uses an intermediate stage called "*view-parameters*". The idea is that the *view-parameters* contain 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's location in the staff, an *x* position relative to the start time, and a *width* value relative to the duration of the event. 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 view-parameters *x*, *y*, and *width* values to draw its two graphic objects from a single set of values.

The ui template class uses two functions to define data-view mappings: *dataToViewParams* which receives the semantic data and returns the view-parameter object, and *viewParamsToData* which performs the opposite mapping. Just as the view-parameters provide a minimal set of variables needed to draw multiple graphic graphic objects from the semantic representation, the *viewParamsToData* function uses the same view-parameters to map back to semantic data. For example, in Figure 5 the mapping only 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, whose position is relative to the parent five-line staff object. Inside the *note*'s *dataToViewParams* and *viewParamsToData* methods, it will need to "ask" its parent objects where to place itself by calling the parent's *childDataToViewParams* and *childViewParamsToData* functions. In deeply hierarchical container structures, it is possible that the parent may need to ask *its* parent for some data as well, and so the parent querying system can provide a way to maintain separation of concerns between different aspects of the notation.

10. IO DEFINITIONS

Running in the server, the io-controller's job is to handle OSC communication with external applications, reading

and writing files, and maintaining the score database. Current default io-definition variables and methods used by the io-controller are:

- *class*: the unique name of the symbol, used to store and lookup the symbol definition.
- *comparator*: a comparator function used in container symbols to sort child symbols. 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. [17]
- *lookup*: called via OSC to look up events at a given value specified by the container (e.g. typically time); returns the query results to the caller via OSC. 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. 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 receive the complete OSC bundle that is sent in, and also have access to the entire score database, and so it is also possible to define multiple ways of looking up (and performing) the score data at the same time; for example multidimensional nearest neighbor lookup, or polytemporal sequencing.

11. CREATING SYMBOLS FROM OSC INPUT

As an illustration of how data is processed through the SYMBOLIST architecture, we can follow the sequence of events in the case of *semantic-to-graphic* mapping; for example when algorithmically generating score data, using an outside process to create new symbols via OSC messages.

By convention, SYMBOLIST uses the DRAWSOCKET message syntax for OSC and JSON interprocess communication, where a "*key*" address is used a keyword to signal which routine should interpret the message, and the "*val*" address contains an object payload (or array of objects) to be processes.

11.1 Symbol Creation from an External Process

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 9.1), 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 7).

A typical drawing command would look something like:

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

Here, we use the JS spread operator "..." to merge the *newView* variable, holding DRAWSOCKET format SVG data organized in three <g> group containers (as described in Section 7), and the HTML dataset information, encoded via the *dataToHTML* helper function into the *val* object with the associated "svg" DRAWSOCKET keyword. The object is then sent to DRAWSOCKET via the *drawsocketInput* helper function to be added to the browser screen.

12. CUSTOM USER METHODS

Users may also create their own custom methods in either ui or io-definition classes and call them from an outside

[17] Pre-sorting increases the efficiency for later lookup queries.

process via OSC (or from other symbol definitions), using the "call" keyword. [18]

Using DRAWSOCKET syntax, the "call" system requires two parameters in the *val* object to lookup and execute the method:

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

However, *all* parameters in the *val* object will be passed to the function, which can be used as a variable length argument when calling the method.

Custom class methods can be used to apply operations to the score or ui, for example a method for 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.

13. CONCLUSIONS

With the new symbol class definition system in place, initial tests seem promising, and support the hope that this new experimental SYMBOLIST implementation will be able to handle a wide variety of score and symbol structures by providing mechanisms for users to compose bidirectional mappings between semantic and graphic representation. The system has the potential to address 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 steps will be to go through the process of developing complete definition libraries for working with different types of notation systems. As a test case, we have 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 of 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, since SYMBOLIST is a graphically oriented authoring environment, it would be convenient if there was a way to create new symbol definitions graphically within the main editor application. To address this issue, further research is planned to develop a GUI for symbol definitions, and other tools to help streamline the process of specifying bidirectional mappings. For example, most mathematical operations have an inverse operation, 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 some issues came up after the Electron version 12 update, which introduced new security measures including context isolation, [19] and increased limitation in using the *require* function used in SYMBOLIST to import user libraries. In order to comply with new Electron web safety measures, SYMBOLIST now uses Webpack [20] to bundle the symbol definitions into a static library file, which is loaded on startup—previously users were able to dynamically load symbol definitions at run time, which seems like a more natural user experience. The new security measures are less than ideal for dynamic updating, however, on the positive side, since SYMBOLIST is now browser-based and uses the same system as DRAWSOCKET for dynamic graphic rendering, SYMBOLIST could now be easily used in networked situations. For example for synchronized score display, or use with multitouch tablets (iPad etc.). In cases where SYMBOLIST is exposed to the internet, the new web-security measures may be important. More testing is needed to determine which features are the most critical, balancing usability with web-security.

For playback/sequencing of SYMBOLIST scores, users can currently send either *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 system is currently implemented in Javascript, which is not the fastest or most temporally precise method of playing back the score. As a starting point we are testing a new Max external called *o.lookup*$\sim$, 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 well for single data sequences (i.e. value of y at point x), however for more robust playback, it might be worthwhile to develop a more complete database lookup system in C/C++, 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, and instrument track selection. It could also be imagined that a score could be exported to playback in a DAW like Ableton Live, and to form connections to other OSC sequencing applications like IRCAM's Antescofo expression language[16].

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

[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.

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

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

[21] https://threejs.org/

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)[17].

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 Max/Gen~ DSP code.

There are 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 it can be learned and used for the creation of new kinds of art.

14. 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] P. Manning, "The oramics machine: From vision to reality," *Organised Sound*, vol. 17, no. 2, pp. 137–147, 2012.

[4] G. Marino, M.-H. Serra, and J.-M. Raczinski, "The upic system: Origins and innovations," *Perspectives of New Music*, pp. 258–269, 1993.

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

[6] ——, "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.

[7] 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.

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

[9] 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.

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

[11] A. Freed, D. DeFilippo, R. Gottfried, J. MacCallum, J. Lubow, D. Razo, and D. Wessel, "o.io: a Unified Communications Framework for Music, Intermedia and Cloud Interaction." in *Proceedings of the International Computer Music Conference (ICMC'14)*, Athens, Greece, 2014.

[12] 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.

[13] 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.

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

[15] 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.

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

[17] 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.

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