SYMBOLIST: BIDIRECTIONAL MAPPING DEVELOPMENTS

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 SYMBOLIST project was developed out of an organic of compositional notation practice.

how much of a background?

In this paper we present a case study for the creation of an open system for graphically developing symbolic notation which can function both as professional quality print or online documentation, as well as a computer performable score in electro-acoustic music and other computer aided contexts. Leveraging Adobe Illustrator's graphic design tools and support for the Scalable Vector Graphics (SVG) file format [1], the study shows that SVG, being based on Extensible Markup Language (XML), can be similarly used as a tree-based container for score information. In the study, OpenSoundControl (OSC) [**?**] serves as middleware used to interpret the SVG representation and finally realize this interpretation in the intended media context (electronic music, spatial audio, sound art, kinetic art, video, etc.). The paper discusses how this interpretive layer is made possible through the separation of visual representation from the act of rendering, and describes details of the current implementation, and outlines future developments for the project.

Initially developed as a method for performing vector graphics, SVG-OSC [**?**]

[1] https://www.w3.org/TR/SVG11/

[**?**]

XML nature of SVG makes it easy to parse and map just like and OSC bundle.

drawing on the OSC research at CNMAT,

just like a piece of paper, SVG could be freely mappable, and *performable* like OSC.

however it can require a lot of specialized code to handle different hierarchy structures and data at different levels in the hierarchy.

the first version of SYMBOLIST was created as a standalone JUCE application Ircam/ZKM [**?**]

1.1 JUCE Version

explain juce version (include in overview?) focused on creating SVG and outputting OSC version (like SVG-OSC approach)

1.2 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

1.3 Drawsocket

[**?**] [**?**]

2. 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 **??** shows a screenshot from the Electron version.

3. IMPLEMENTATIONS

Electron [2] version (node + chrome w/ special electron IPC)
Max version (node + jweb + Max patch for IPC)

3.1 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

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

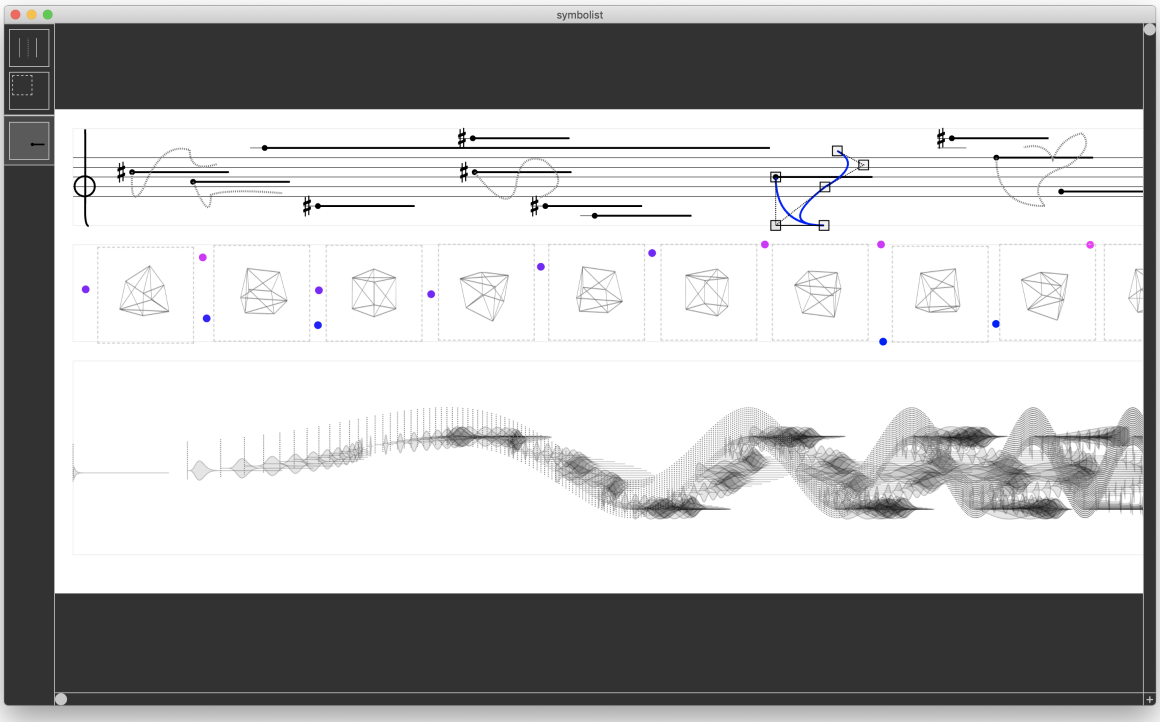

Figure 1. SYMBOLIST screenshot, showing some different types of staves, and editing capabilities.

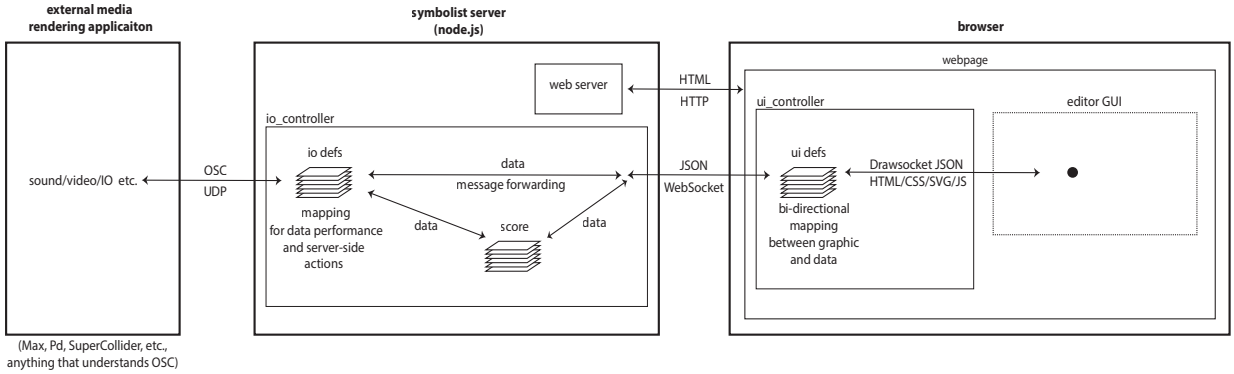

Figure 2. SYMBOLIST architecture.

through graphic interaction. The *ui_controller* runs in the browser (see Figure **??**), 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:

    - 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, and maintains the central *score* database in its semantic data format (described below).

4. GRAPHICAL AUTHORING AND INTERACTION

The graphic user interface of SYMBOLIST (Figure **??**) 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.

4.1 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 **??**) 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.

4.2 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 **??** below).

4.3 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.

5. DATA REPRESENTATION

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

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.

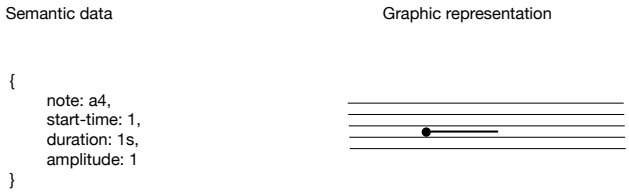

Figure 3. *data* vs *graphic* representation of the same information.

6. 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.

7. SEMANTIC DATA FORMAT

Within the SYMBOLIST application, semantic data is stored as javascript objects, and read/written in JSON format [3], 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).

In addition to the required *id* and *class* attributes, symbol objects may include any number of other *attributes*[4] of the symbol (*pitch*, *amplitude*, etc.). For example a simple semantic object written in JSON might look like:

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

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

```
{
   "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.

7.1 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.

8. 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:

```
{
   "about" : "symbolist will read a json file to configure
      the palette setup, this can be used to dynamically
      change the application layout and tools",
   "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.

9. 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.

9.1 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.

9.2 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>
```

9.3 dataset-elements

Since SYMBOLIST is constantly mapping to and from semantic data and its graphic representation, we are using the HTML *dataset* feature [5] to store the semantic data inside the top-level *symbol* element (<g> or). The HTML dataset attributes use the prefix "*data-*". [6]

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>
```

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

[6] 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.

10. 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 [**?**] could be used to guide the movement of robotic motors, create haptic feedback for live performance [**?**], and so on.

SYMBOLIST provides several different options for sorting and looking up data (see Figure **??**), 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.

11. 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 **??**) 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 [**?**, **?**, **?**, **?**, **?**].

Graphic representation to semantic data mapping (Figure **??**) 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 **??**) 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 **??** for further discussion.

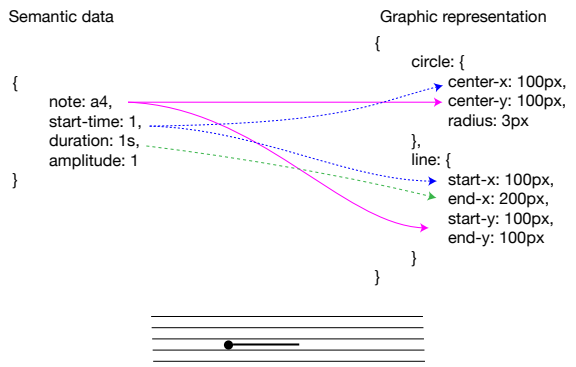

Figure 4. *semantic data* mapped to create a *graphic* representation from input data.

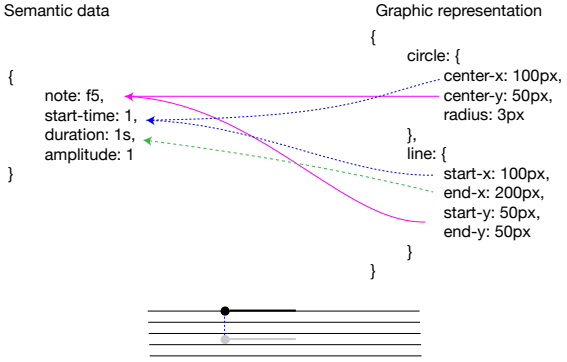

Figure 5. If edited graphically, the updated graphic data is then mapped back to *semantic data* representation.

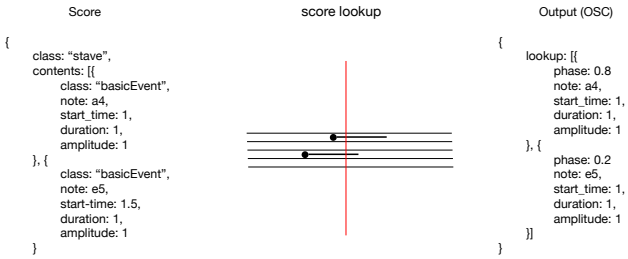

Figure 6. Using the lookup method defined by the symbol class, the *semantic data* can be used to perform external instruments via Open Sound Control.

12. SYMBOL DEFINITIONS

Symbols are defined as Javascript classes which are stored and recalled when symbol actions are performed. Through creating definitions, users are able to form libraries of symbols that can be used together to fit the tools and representation to address a given use-case scenario. Definitions define the mapping relationships between the symbol's *semantic*, *graphic*, and output *performance* data.

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 [7] (Figure **??**), 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, [8] 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 **??**). 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.) [9] 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.

12.1 UI Definitions

Variables and methods called by the *ui_controller* are:

- *class*: the unique name of the symbol.
- *palette*: used for container symbols, an array of class names of other symbols that can be used within this container.
- *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.

12.1.1 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 **??** and **??** 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 **??** the *note* property needs to

[7] and potentially separate devices.
[8] https://developer.mozilla.org/en-US/docs/Glossary/MVC
[9] https://developer.mozilla.org/en-US/docs/Web/API/Document

be mapped to a pixel position that is used for both the center point of a graphic circle (note-head) as well as the starting point for a line (duration indication). In reverse, Figure **??** 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 one-to-many data to graphic situations.

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 **??** 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 **??**, 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 methods to coordinate child objects with parent containers: *childDataToViewParams* and *childViewParamsToData*. For example, in Figures **??** 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.

12.2 IO Definitions

Variables and methods called by the *io_controller* are:

- *class* (string) class name, corresponding to class name in UI Definition.
- *comparator* (a,b)=> comparator function to use to sort this symbol type, return -1, 0, or 1
- *lookup* (params, obj_ref) => hit detection function called when looking up from query point, returns information to send back to caller. ‘params‘ are user parameters included in the lookup query, ‘obj_ref‘ is the instance of this class type. When the ‘lookup‘ key is received by the ‘io controller‘, the system looks up the element by its ‘id‘, generally the id will be of a container object. Then it is up to the container object's ‘lookup‘ function to iterate its child objects and accumulate them into an output array to send back to the calling application.

12.3 Custom User Methods

Users may also create their own methods and call them

- *call*: calls a function in the one or both of the class definitions. All of the parameters in the *val* object will be passed to the function as an argument. Return values from the *io* controller are with the tag *return/io* and *return/ui* from the *ui*.
  - *class* (required) class of the object to call
  - *method* (required) name of object function to call

```
{
   /key : "call",
   /val : {
      /function : "functionName",
      /class : "className",
      /id : "foo"
      /someValue : 1,
      /anotherValue : 2
   }
}
```

Note that the system will pass the same call to both definitions, so if both have a function of the same name they will both be called. * *drawsocket*: forwards DRAWSOCKET format messages directly to DRAWSOCKET, bypassing the symbolist mapping.

13. SYMBOL DEFINITION FUNCTIONAL EXAMPLES

13.1 Creating Symbols from OSC input

The simplest example of mapping definition is the *semantic to graphic* mapping definition.

13.1.1 OSC is received by the io_controller

The *io_controller* has a set of messages that it understands which can be used for a wide range of actions. SYMBOLIST uses the DRAWSOCKET *key/val* message syntax, so each OSC bundle should be formatted with a *key* address, which is a keyword flag to signal which routine should interpret the message in the *io_controller*'s *io_receive* function; while the *val* address contains an object (or array of objects) to be processed.

And OSC message with the *data* keyword can be used to add new semantic data objects to the score-model, and create new graphic symbols in the editor from an external process; for example by generating scores through algorithmic processes, via textual authoring, via mapping from gestural controller, etc.

Upon receiving an OSC message with a *data* flag, the object attached to the *val* address are added to the model, and then relayed to the *ui_controller*. In the *ui_controller*, the semantic data is mapped to graphic data using the symbol's *class* definition. The class's name provides a mechanism to lookup the class definition in both controller scripts.

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* key has a few required and optional attributes:

- *class*: (required) the class type of the object to create.
- *container*: (required) the *id* of the container symbol class to put the object in.
- *id*: (optional) an id to use for the data object, if non is specified a (long) unique string will be generated.
- Other required or optional parameters will depend on the symbol definition.

13.1.2 Data to View Mapping in the ui_controller

fromData

If an OSC message is received containing data to create a new symbol, the *ui_controller* calls the object's 'fromData' function, which maps from the data representation to the graphic drawing commands. The 'fromData' function should: 1. send the drawing commands to the browser display (via DRAWSOCKET usually, using the 'drawsocketInput' API method). Include the data content into the symbol by using the HTML 'dataset' (you can use 'ui_api.dataToHTML(data' helper function to create the 'data-' tags)

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

13.2 Creating Data from Graphic Interaction

briefly describe the sequences of events

createNewFromMouseEvent (template function)

Working from the opposite direction, users can create semantic data entries in the score through graphic interaction. The process for achieving this is a bit more complicated than the

14. TEMP

SYMBOLIST has built in handlers for a set of messages received via OSC, which can be extended by user scripts, using a key/val syntax, where the *key* specifies the function to call, and the *val* are the parameter values to use for the call.

For example, here is a *lookup* query to find elements that are returned by the parameters *time* in the container with the *id* "trio".

```
{
   /key : "lookup",
   /val : {
      /time : 0.1,
      /id : "trio"
   }
}
```

The OSC message API supports the following keys:

- *data*: adds a data object to the score, and sends to the ui to be mapped to graphical representation. Parameters include:
  - *class* (required) the class type of the object to create
  - *container* (required) the container symbol class to put the object in (in case there are multiple containers that support the same symbol type)
  - *id* (optional) an id to use for the data object, if non is specified a (long) unique string will be generated.
  - Other required or optional parameters will depend on the symbol definition.

```
{
   /key : "data",
   /val : {
      /class : "fiveLineStaveEvent",
      /id : "foo"
      /container : "oboe",
      /time : 0.13622,
      /ratio : "7/4",
      /duration : 0.1,
      /amp : 1
   }
 }
```

- *lookup*: looks up a point in a container, based on a sorting function specified in the definition. For example, this can be used to get all events active at a given time. Parameters:

  - *id*: (required) the *id* of the container to lookup in. Containers will generally iterate all child objects, so for example if you use the *id* of the top level score you should be looking up in to all sub-containers.

- *getFormattedLookup*: optional function that might be defined in an *io* script that outputs an object formatted for a different type of player/render. For example, this function might return a list of */x* and */y* values for use with the *o.lookup* Max object, or create a MIDI file export etc. All parameters included in the *val* object will be sent to the *getFormattedLookup* as a parameters object. Parameters:

  - *id*: (required)

- *call*: calls a function in the one or both of the class definitions. All of the parameters in the *val* object will be passed to the function as an argument. Return values from the *io* controller are with the tag *return/io* and *return/ui* from the *ui*.

  - *class* (required) class of the object to call
  - *method* (required) name of object function to call

```
{
    /key : "call",
    /val : {
        /function : "functionName",
        /class : "className",
        /id : "foo"
        /someValue : 1,
        /anotherValue : 2
    }
}
```

Note that the system will pass the same call to both definitions, so if both have a function of the same name they will both be called. * *drawsocket*: forwards DRAWSOCKET format messages directly to DRAWSOCKET, bypassing the symbolist mapping.

15. CONCLUSIONS AND FUTURE DEVELOPMENTS

known issues - unable to dynamically refresh definitions in the browser using the require function after Electron v12, using Context Isolation [10] as suggested to address possible security issues, currently webpack is used to bundle the definitions ¿¿ need to look into what is possible, since for user defined scripts we definitely will want to be able to tweak them without running webpack again.

continue work on CP notation GUI for definitions C/C++ score player, optimized for fast lookup - write to buffer? further experiments in representation, spatial audio, eurorack modules, connect with antescofo? connect with faust? create audio code via faust, or maybe pd? networked display via DRAWSOCKET compose on tablets with pen?

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

Acknowledgments

thanks james! thanks georg! thanks jean!

16. REFERENCES

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

[2] R. Gottfried, "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.

[3] 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.

[4] 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.

[5] 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.

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

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

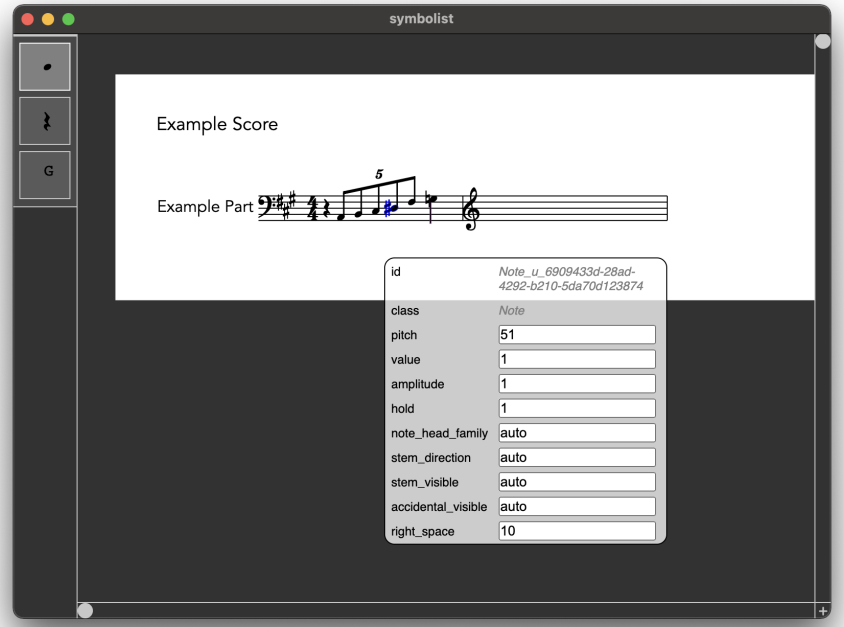

Figure 7. SYMBOLIST screenshot, showing some different types of staves, and editing capabilities.

[12] 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.

[13] 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.