Fun with Interfaces (SVG Interfaces for Musical Expression)

Benedict R. Gaster and Nathan Renney
Computer Science Research Centre (CSRC)
Department of Computer Science and Creative Technology
University of West of England
Bristol, UK
(benedict.gaster,nathan.renney)@uwe.ac.uk

Carinna Parraman
Centre for Fine Print Research (CFPR)
Department of Creative Industries and Education
University of West of England
Bristol, UK
Carinna.Parraman@uwe.ac.uk

Abstract

In this paper we address the design and implementation of custom controller interfaces, bridging the issue of user mapping between action and sound in interactive music systems. A simple framework utilizing functional specifications for musical interfaces and their mappings is presented, in terms of a subset of Scalable Vector Graphics (SVG); interfaces can be described using a simple Haskell based 'controller DSL' or equally using a vector drawing application (i.e. Illustrator).

We demonstrate the practical use of our system for specifying interfaces as SVGs combined with Faust, a functional DSL for Digital Signal Processing (DSP), in the context of building digital musical instruments. We combine these into a hardware and software audio toolkit, with synthesizers, a sampler, effects, and sequencers. Written in the systems programming language Rust, it demonstrates utilizing the output of our DSLs, providing a type safe and high-level framework for DSP and interface development, with the performance benefits of Rust. Working examples of custom interfaces are described, using ROLI's Lightpad and Sensel's Morph.

CCS Concepts • **Applied computing → Sound and music computing**; • **Software and its engineering → Functional languages**;

Keywords Haskell, Rust, SVG, DSL, DMI

ACM Reference Format:
Benedict R. Gaster and Nathan Renney and Carinna Parraman. 2019. Fun with Interfaces (SVG Interfaces for Musical Expression). In *Proceedings of the 7th ACM SIGPLAN International Workshop on Functional Art, Music, Modeling, and Design (FARM '19), August 23, 2019, Berlin, Germany*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3331543.3342579

FARM '19, August 23, 2019, Berlin, Germany

ACM ISBN 978-1-4503-6811-7/19/08…$15.00
https://doi.org/10.1145/3331543.3342579

1 Introduction

Controller and gesture interaction with audio and/or visual media is today ubiquitous, requiring the development of intuitive software solutions for interaction design. Designing and building these interfaces often require extensive domain expertise in audio and visual media creation, e.g. the musician, but additionally in engineering and software development. In this paper we focus on custom controller-based interactive systems for sound and musical performance, with a focus on an intuitive and simple design process that is accessible to artists.

Our particular take is on the controller for these systems, proposing a simple framework for describing them, combining interface descriptions specified using Scalable Vector Graphics (SVG) [20] and Open Sound Control (OSC) [8]. Unlike authors such as Bongers and Jorda [3, 12], who look at the Digital Musical Instrument (DMI), its interface and sound generating engine, holistically, in this work we instead isolate these two components in order to more completely explore the controller. In doing so, it allows this work to focus on the design, specification, and implementation of a DMI interface, while utilizing a variety of off the shelf and custom sound engines.

What is a musical interface? To answer this question, we first consider the more general question, what is Digital Musical Instrument? Following Miranda and Wanderley [15], Figure 1 defines the essence of what might be considered a DMI. The division between the gestural interface and the sound engine is bridged by what Magnusson [13] calls the mapping engine, here defined within the instrumental model as a core component of the instrument itself, while isolating the controller.

Below we give a brief outline of these three components, colouring the discussion slightly by splitting the controller into interface control and interface layout.

As conceptualized in the side image below, an audio engine is a set of functions to process and generate sound, whose goal is to produce one or more channels of audible output.

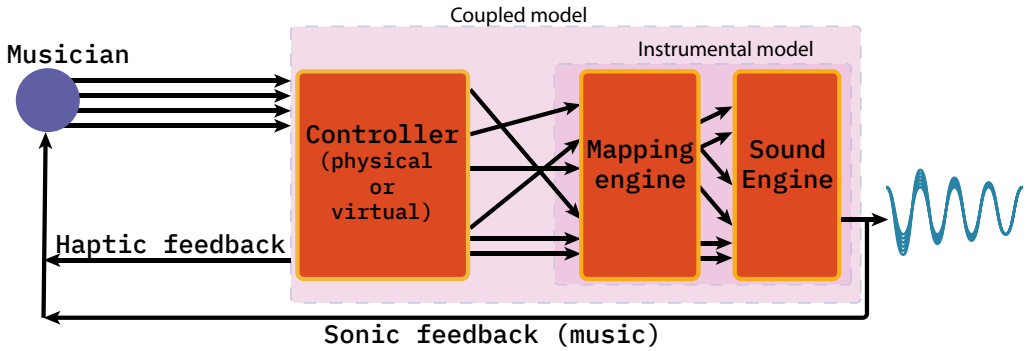
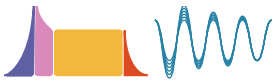
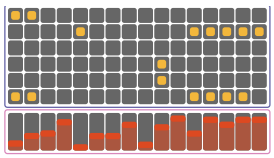


Figure (1) Model of a Digital Musical Instrument, showing the separation between gestural controller and sound production.



An audio engine runs at audio rate, e.g. 44,100Hz, but is often modulated at control rate, commonly in the range of 1 to a few hundred Hz. Modulation might be internal to the audio engine itself, e.g. sweeping the cut off frequency of a low pass filter using an LFO, or directly controlled via user input, e.g. a hardware slider to control the master volume of a mixer. In general, the association between input sensor and modulated value is defined by the user and sensors and controlled via the mapping engine, which may additionally provide the ability to select or even define transfer functions mapping one or more input values to one or more output control values. Mapping might range from simple linear functions, e.g. frequency control, to complex history based functions, e.g. utilizing neural networks for gesture processing.



As abstracted by the side image, a user controls an instrument through a physical/virtual interface that in turn produces control messages, sent to the mapping engine, via an agreed communication protocol, e.g. USB or UDP.

In general, an interface might consist of any form of input sensor, however, for this work we consider a more restricted palette based on common off the shelf controllers, such as Ableton's Push 2, for musical instruments, including touch pads, sliders, potometers, buttons, and so on.

As conceptualized by the side image, the layout of an interface's control components can be varied and many design principles are at play from their initial conception to physical (or virtual) manifestation.

It is not always clear where the interface ends and the sound engine begins, for example, the Fiddle whose interface, its strings, directly plays a physical role in the instrument's timbre. For the purpose of this paper we choose to consider a clear separation between the control interface and the sound engine. Furthermore, as shown diagrammatically in Figure 1, the mapping engine can be considered part of the sound engine and is independent of the proposed framework.

In this paper we propose and demonstrate a simple functional framework for describing custom controllers that utilize Scalable Vector Graphics (SVG) XML as a Domain Specific Language (DSL) for interface descriptions. Custom SVG attributes enable semantic details of interfaces to be described in a portable way, enabling interfaces to be initially developed within an application such as Adobe Illustrator and later described in a Haskell based interface DSL to supply semantic information, such as control messages and functions. An overview of our approach, with examples, is given in Section 2.

Throughout this paper we demonstrate the practical use of SVG interfaces in the context of the design and implementation of DMIs, through the audio application Muses that we have developed. Muses is a hardware and software application/toolkit, with synthesizers, a sampler, effects, and multiple sequencers. Written in the systems programming language Rust [2], it demonstrates the use of SVG interfaces and additionally, utilizes the DSP functional programming language Faust [16], providing a type safe and high-level framework for DSP and interface development, with the performance benefits of Rust.

We establish the utility of SVG interfaces to controllers themselves by building a number of practical custom interfaces using Sensel's Morph, a highly sensitive touch sensor with both x, y, and pressure readings communicated over USB. Additionally, we target ROLI's Ligthpad Block[1], also a touch and pressure based sensor, demonstrating the direct compilation of SVG interfaces to Lightpad's interface programming language, Littlefoot[2]. We also show interfaces

[1] https://roli.com/products/blocks/lightpad-m
[2] https://github.com/WeAreROLI/JUCE/blob/master/modules/juce_blocks_basics/littlefoot/LittleFoot%20Language%20README.txt

derived from an Implicit CAD[3] backend for the system, but due to being early in the development cycle do not present the details.

The remainder of this paper is structured as follows:

- Section 2 provides an overview of the proposed system for SVG interfaces;
- Section 3 details our approach to SVG interfaces, including their specification as a Haskell based DSL, representation as standard SVGs, and compilation to an intermediate JSON format that can be consumed by different backends, i.e. hardware platforms for the interfaces themselves;
- Section 4 implements our ideas in the Muses audio system, utilizing two external control surfaces, the Sensel Morph and ROLI's Lightpad, as targets for SVG interfaces; and finally
- Section 5 discusses related work and concludes with pointers to possible fruitful directions for the development of SVG interfaces and the audio applications they are intended to control.

More details about the project, our implementation, and examples are publicly available from https://muses-dmi.github.io/.

2 Overview

We will provide a short overview of SVG interfaces by considering the MPC style interface shown in Figure (a) 2. The image shown in Figure (b) 2 is a realization of the SVG interface using Swell (Braille) paper to provide a tactile playing surface. The swell print can be placed directly on a touch based interface, such as Sensel's Morph whose implementation is described in Section 4. In this section we will outline how a user can develop such an interface with the proposed SVG system for interfaces. For simplicity we consider just the stop button, captured using its standard icon: a square in between the play (triangle) and record (circle) in the top right of both images in Figure 2.

2.1 A Little Programming Language

Interfaces are described using a controller DSL that is a strongly typed functional language for an extended subset of SVG, which includes structures for standard controller types such as play and record, and more general ones such as pads and sliders. Our stop button can be expressed as:

```
stop ! #x 10 ! #y 10 ! #size 3
```

The arguments x, y, size are required and like SVG attributes are named. In this case using the notation, #name, to specify an argument's name, followed by the argument itself, in this case 10. As per SVG arguments are ordered and multiple arguments are combined with the combinator !. The above code describes a stop button whose top left corner is (10,10) and a width and height of 3. Implicit in its definition is the fact that interacting with a stop button, i.e. pressing it, will cause it to output a "/stop/" message.

More generally a stop button is an instance a pad and could just as easily be defined as such:

```
pad ! #x 10 ! #y 10 ! #size 3
    ! #address "/stop/"
```

Here the additional argument address must be specified, which in this case is the button's "/stop/" message, but more generally can be any Open Sound Control (OSC) message supported by the intended receiver.

Controllers can be composed together to form interfaces. For example, the following describes not only the stop button from Figure 2, but play and record too:

```
stop ! #x 167 ! #y 20 ! #size 25
<>
play ! #v1 (162,32) ! #v2 (142,20)
     ! #v3 (142,45)
<>
record ! #x 209 ! #y 32 ! #r 12
```

Interfaces form a monoid [18] and the combinator <> is Haskell's standard Semigroup associative operator.

2.2 SVG Representation

An interface is represented as an SVG, for example, the stop button above is compiled to an underlying SVG rectangle and could have been defined more explicitly as:

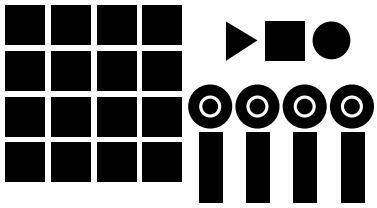
```
rect 10 10 3 3 # itype ipad
     # iaddress "/stop/"
```

However, at this point much of the syntactic sugar has been removed, for example, named arguments are now explicit and order matters. Additionally, the controller's type (ipad) and message address must be explicitly defined, as an SVG rect has no knowledge of this additional semantic information. This example highlights the use of a restricted form of reverse function application (#) to 'apply' non-standard SVG attributes.

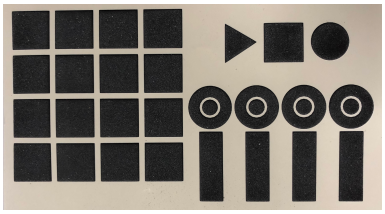
2.3 Compilation

Interfaces are designed, shared, and modified as SVGs, either directly or using serialization and deserialization to and from our programmatic interface DSL. However, while SVGs interfaces can be printed they do not provide an execution model per say. For this an SVG compilation model is defined, enabling interfaces to be mapped to a variety of touch based control surfaces. For example, the SVG interface captured diagrammatically in Figure (a) 2 can be mapped directly to Sensel's Morph touch interface. The Morph provides the ability to lay a tactile materialization of an interface, in silicon for example or as shown in Figure (b) 2 printed on Swell paper, directly down on to its touch surface. In real time it

[3] https://github.com/colah/ImplicitCAD



(a) SVG Representation of MPC style interface.



(b) Tactile MPC Style interface printed on Swell (Braille) paper.

Figure (2) Outputs of SVG (rendered) and Swell Paper.

generates a pressure image of interactions that can be processed via USB. ROLI's Lightpad on the other hand is a small pressure based interface that supports execution of Littlefoot DSL programs directly on the device itself. An example, SVG interface for the Lightpad is shown in Figure 3,

Any execution model for SVG interfaces must be able to support mapping directly to the Morph and Lightpad, along with other interfaces, such as an iPad. For this we propose and implement a compilation pipeline based on tessellation and a human readable intermediate language (IR) based on JSON.

3 SVG Interfaces

In the previous section, we provided a short overview of some basic primitives for building SVG interfaces and discussed their compilation to an intermediate representation suitable for execution on a number of touch devices. In this section, we describe SVG interfaces, how they can be defined in terms of a controller based DSL embedded in the functional programming language Haskell and a compilation approach that enables real time execution of a variety hardware platforms. All the time our interface design enables the creation of tactile, physical interfaces that can be produced directly from the SVG representation, independently of the compilation method.

3.1 A Controller DSL

The core elements of the DSL are given in Figure 5. For the most part the DSL itself is implemented in standard Haskell, as defined by Marlow [14], however, to fit closer with SVG's named attributes it uses a variant of the Haskell package Named arguments[4]. Named is a lightweight library for named function parameters based on overloaded labels. Like SVG attributes named arguments can be specified in any order and additionally can provide call-site documentation, which we have found useful when non-programmers develop custom interfaces.

As values, named parameters take the form `! #x 123`, where `!` indicates the use of a named argument, `x` is the name of the argument (with `#` indicating a first class label within the `!` expression) and `123` is the argument's value. For types, the type constructor `:!` takes a label, represented as a type promoted string, and the argument's type, for example, the expression `! #y 234` has type `"x" :! Int`.

With the exception of the endless controller, the domain specific and global OSC address controllers are expressed semantically in terms of the SVG shape controllers. These three core controllers map directly to the corresponding SVG shapes. For example, consider the expression:

```
stop ! #x 167 ! #y 20 ! #size 25
```

which describes a stop button controller object, with its right most corner placed at position (167, 20) and with a width and height of 25 (mm). When compiled it becomes the following SVG shape element:

```
<rect inter_osc_address="/stop"
  height="25" width="25"
  inter_type="pad" inter_osc_args=""
  x="167" y="20"/>
```

The SVG shape `rect` has a number of standard attributes, including "x", "y", "width", and "height", along with generic shape attributes such as "fill", "stroke", etc. not specified in this example, but it also uses a selection of interface custom attributes, e.g. `inter_osc_address`. The meaning of the standard attributes are unchanged from the SVG specification and are not considered further, instead we focus attention on the custom attributes necessary for a controller to be well defined.

Table 1 defines the domain specific attributes for controllers. The first attribute reflects the kind of controller it

Attributes	Description
inter_type	Controller type (pad, vslider, etc.)
inter_osc_address	OSC address
inter_osc_args	OSC static arguments (int or float)
min	min value for slider, endless
max	max value for slider, endless
init	initial value for slider, endless
incr	increment for slider, endless

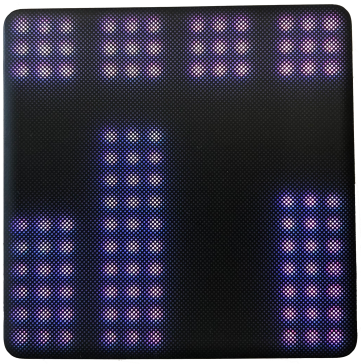
Table (1) Domain Attributes for Controllers.

[4] https://github.com/monadfix/named.

```
lightpad =
    mconcat (map p [(0,100), (4, 101), (8, 102), (12, 103)])
<>
mconcat (map s [(0,104), (4, 105), (8, 106), (12, 107)])
where
  p = \(x, cc) -> pad ! #x x ! #y 0 ! #size 3 ! #address "/midicc"
                      # iargs [cc]  # fill "rgb(217,137,188)"
  s = \(x, cc) -> vertSlider ! #x x ! #y 4
                    ! #width 3 ! #height 11 ! #min 0 ! #max 127 ! #address "/midicc"
                      # fill "rgb(96,95,164)" # iargs [cc]
```

Figure (3) SVG DSL Example (4 pads and 4 sliders).

(a) SVG Representation of interface from Figure 3.



(b) Interface in action, loaded on a ROLI Block.

Figure (4) Outputs of SVG (rendered) and Littlefoot (running on Block).

represents. For example, a vertical slider is represented by a slider of type inter_type="vslider", while the above stop button is of type inter_type="pad". The next two attributes define the type of control messages produced by a given controller. As discussed in Section 2 interfaces communicate to the outside world using Open Sound Control (OSC).

OSC messages are typically transported via the internet and within local subnets using UDP/IP, although for this work we have focused on communication limited to same machine, i.e. the loop back device. OSC messages consist of an address pattern, a type tag string, 0 - n arguments, and an optional time tag. Similar to URLs or Unix filenames, address patterns form a hierarchical name space. Type tag strings are a compact string representation of the argument types, while arguments are represented in binary form and include 32-bit two's complement signed integers and 32-bit IEEE floating point numbers[5].

In general, OSC messages have arbitrary addresses and often describe a path where the message should be routed in the receiver application, e.g. /sequencer/mute might indicate that the message is for a sound engine's sequencer to mute a particular channel. The actual channel could be specified in the address, but rather is passed as one of the message's arguments. A complete message instructing the audio engine to mute channel 2 might look like[6]:

```
/sequencer/mute 1
```

In general a controller's OSC address and static arguments are defined by the user, but in some cases the SVG DSL provides a small number of predefined global messages, as shown in Figure 5 and the stop example above. Controllers such as the one imagined above to send mute messages have only static arguments, i.e. the channel being muted is known during the specification of the controller, in general, this is not the case. For example, consider a slider to control the volume of channel 0 in a mixer component, expressed as:

```
vertSlider #x (125 + chan * 30) ! #y 90
 ! #width 15 ! #height 45
 ! #min 0 ! #max 127
 ! #address "/mixer/volume" # args [0]
```

The address is the path to the mixer's volume control and the single static argument specifies the particular channel

[5]OSC arguments can also include null terminated arrays and binary blobs of data, however, they are not currently utilized in our system.

[6]Assuming channels are indexed from 0.

```
-- svg shape controllers
rect :: "x":!Int -> "y":!Int -> "width":!Int -> "height":!Int -> Controller
circle :: "x" :! Int -> "y" :! Int -> "r" :! Int -> Controller
polygon :: "points" :! [(Int,Int)] -> Controller

-- domain specific controllers
pad :: "x":!Int -> "y":!Int -> "size":!Int -> "address":!Text -> Controller
toggle :: "x":!Int -> "y":!Int -> "size":!Int -> "address":!Text -> Controller
horzSlider :: "x" :! Int -> "y" :! Int -> "width" :! Int -> "height" :! Int ->
                "min" :! Int -> "max" :! Int -> "address" :! Text -> Controller
vertSlider :: "x" :! Int -> "y" :! Int -> "width" :! Int -> "height" :! Int ->
                "min" :! Int -> "max" :! Int -> "address" :! Text -> Controller
ciPad :: "x":!Int -> "y":!Int -> "r":!Int -> "address":!Text -> Controller
endless :: "cx" :! Int -> "cy" :! Int -> "or" :! Int -> "ir" :! Int ->
             "address" :! Text -> Controller

-- controllers with global OSC addresses
stop :: "x" :! Int -> "y" :! Int -> "size" :! Int -> Controller
record :: "x" :! Int -> "y" :! Int -> "r" :! Int -> Controller
play :: "v1" :! (Int,Int) -> "v2" :! (Int,Int) -> "v3" :! (Int,Int) -> Controller

-- attributes
border :: Controller -> Controller
fill :: RGB -> Controller -> Controller
imin :: Int -> Controller -> Controller
imax :: Int -> Controller -> Controller
```

Figure (5) SVG DSL

number, but the actual value of the slider cannot be specified statically. It is only known when it is initialized[7] and when the user adjusts the value by interacting with the controller physically. In this case the behaviour of a slider, when physically interacted with, causes it not only to emit static arguments, but also to include the slider's position value, in this case a number between 0 (#min 0) and 127 (#max 127).

3.2 Intermediate Compilation

Individual devices intended as targets for an SVG interface can differ considerably with respect to how they materialize an interface in a practice. For example, the Sensel Morph, detailed in Section 4.1, provides a C API for reading contact information from the device with USB, while ROLI's Lightpad, whose implementation is described in Section 4.2, works as an embedded device capable of executing programs described in the C like DSL, Littefoot. While SVGs provide an interesting and practical approach to describing interfaces they offer some challenges when mapping them to devices themselves. In particular, while describing controllers in terms of 2D vectors, combined with domain specific attributes, provides an accessible design methodology for humans, it can come at a cost for a device driver, e.g. for the Sensel, and is even more of an issue for resource constrained embedded devices, such as ROLI's block. The driver or embedded code must provide a high performance path from human gesture, i.e. interacting with the physical interface, and output messages.

In the remainder of this section we outline an intermediate representation, generated automatically from interface SVGs, which enables straight forward implementation in driver code and compilation to embedded DSLs, such as ROLI's Littlefoot, with little run time performance impact. In particular, the intermediate representation removes the need for complex collision detection of gesture to action. Of course, our representation does not prevent complex transfer functions, i.e. the interpretation of a gesture to an action, which in general can be arbitrarily complex. Although the current set of supported sliders, pads, and so on do not introduce much computational complexity, there is nothing in our approach that prevents more complex controller transfer functions being added.

[7] An initial value can be specified using the attribute # init , otherwise it is assumed to be zero.

JSON is used as an intermediate representation for interfaces, that can either be loaded directly by a device backend, see the Sensel Morph example in Section 4.1, or an additional translation step can be performed to produce a representation suitable for upload on to hardware, see the ROLI Lightpad implementation described in Section 4.2 for an example of this approach. Alternatives to JSON are likely to be just as suitable for our needs, however, JSON was chosen as it is a simple open standard, in wide use, and is human readable.

The IR itself is specified as a JSON schema[8], emitted here for space, but can be found on the project's documentation pages[9]. As an example consider the SVG interface specified in Figure 3 and shown in Figure 4. The following JSON, cut down for space, is outputted via our SVG compiler:

```
{
    "controllers" : [
        {
            "type_id" : "pad",
            "id" : 1,
            "rgb" : "rgb(217,137,188)",
            "args" : [
                100
            ],
            "address" : "/midicc"
        },
        ...
        ],
    "buffer" : [
        [
            1,1,1,0,2,2,2,0,3,3,3,0,4,4,4,
            1,1,1,0,2,2,2,0,3,3,3,0,4,4,4,
            1,1,1,0,2,2,2,0,3,3,3,0,4,4,4,
            0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
            5,5,5,0,6,6,6,0,7,7,7,0,8,8,8,
            5,5,5,0,6,6,6,0,7,7,7,0,8,8,8,
            5,5,5,0,6,6,6,0,7,7,7,0,8,8,8,
            5,5,5,0,6,6,6,0,7,7,7,0,8,8,8,
            5,5,5,0,6,6,6,0,7,7,7,0,8,8,8,
            5,5,5,0,6,6,6,0,7,7,7,0,8,8,8,
            5,5,5,0,6,6,6,0,7,7,7,0,8,8,8,
            5,5,5,0,6,6,6,0,7,7,7,0,8,8,8,
            5,5,5,0,6,6,6,0,7,7,7,0,8,8,8,
            5,5,5,0,6,6,6,0,7,7,7,0,8,8,8,
            5,5,5,0,6,6,6,0,7,7,7,0,8,8,8,
        ]
    ],
    "interface" : "lightpad"
}
```

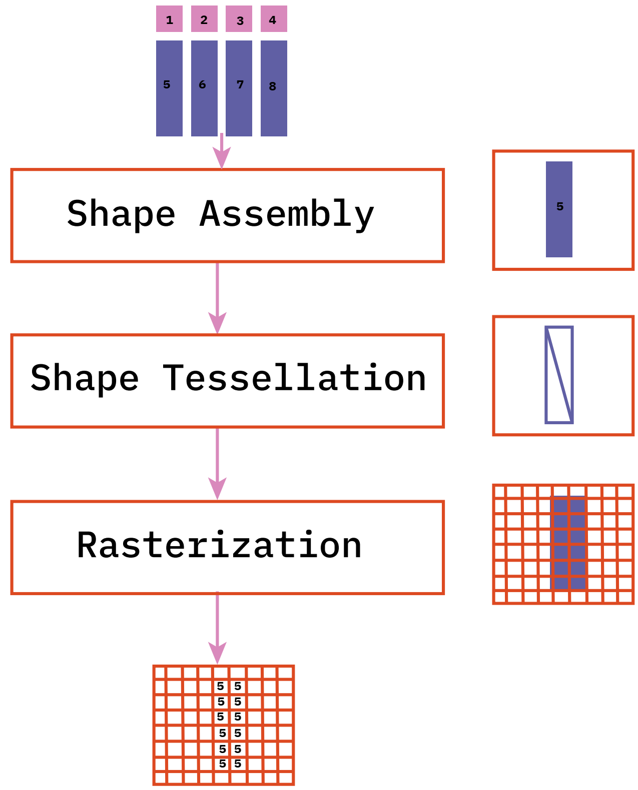


Figure (6) Interface Rasterization Pipeline

The controllers section contains a mapping from the SVG attributes for a controller, with a unique identifier assigned to each one. This identifier appears in the buffer array, which represents a 2-dimensional map from (x,y) touch events produced by the hardware controller to a control's identifier. The buffer is a one to one mapping with the intended device, for example, here the 2-dimensional space is 15x15, representing the touch sensor array on ROLI's Lightpad. Further details of the table and how it is produced from SVG shape descriptions is the topic of the following sub-section.

3.2.1 Tessellation

As discussed above, a touch interface is a transfer function between user input (touch), captured as one or more (x,y) positions in 2-dimensional Euclidean space, to an action, e.g. sending a control message. In general, such interactions would require complex collision detection, however, taking inspiration from the process of rendering an SVG on a screen, we note that it is possible to describe the intersection problem as a function from 2D coordinates, the touch event, to a point within a shape or path describing the controller. Once the shape and this controller is uniquely determined, then an action's function is by definition also determined.

[8] http://json-schema.org/draft-07/schema#

[9] https://github.com/muses-dmi/svg-creator/blob/master/docs/interfaces.md

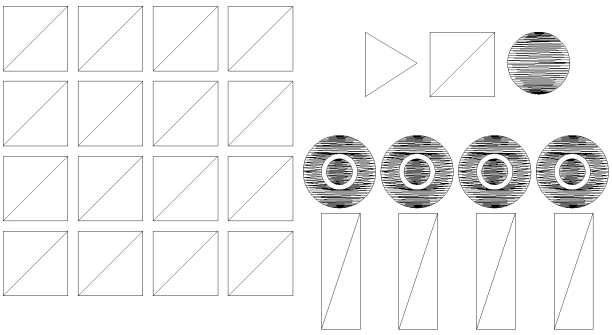


Figure (7) Tessellation of MPC style interface

The function in question, from a 2-dimensional vector description of a shape, simple (e.g. SVG rect) or complex (e.g. SVG Bezier curves and paths [9]), can be described by the pipeline shown in Figure 6.

The function is the well known and is simply the process of rasterization [1], but rather than storing colours in the resulting "image", a unique ID for the controller is written out instead. Unlike general collision detection the resulting function has O(1) complexity.

The algorithm iterates over an interface's set of shapes and for each shape first preforms Shape Assembly, which among other things assigns a unique identifier; the resulting shape is then tessellated into simple triangles; and finally the resulting set of triangles are rasterized with each "pixel" within the triangle mapped to the corresponding position in the output bitmap, storing the shapes unique identifier. Rasterization is the most complex phase and simple interfaces, particularly ones containing non convex-polygons, can lead to a large number of resulting triangles. For example, the MPC style interface given in Figure 2 generates the triangle mesh shown in Figure 7, which contains 2862 vertices using our Rust based tessellation algorithm[10].

Given a triangle mesh rasterization is implemented using the standard half-space algorithm [17], common to graphics APIs such as DirectX[11] and Vulkan[12]. Unlike these graphics APIs our rasterization algorithm does not utilize optimization techniques such as depth buffering or front face culling. Moreover, the pipeline presented in Figure 6 is implemented completely in software and GPU acceleration was found to only complicate the implementation and provided no additional performance benefits. While interfaces can contain many controllers, in the range of 10-100, the number of resulting triangles is very small when compared to even simple 2D/3D graphics scenes and furthermore the compilation of interfaces is offline and thus is not a performance bottleneck.

[10]The Rust tessellation algorithm is based on the Muses' 2D renderer, which it turn is implemented with the Rust library Lyon.

[11]https://docs.microsoft.com/en-us/windows/desktop/direct3d12/what-is-directx-12-

[12]https://www.khronos.org/vulkan/

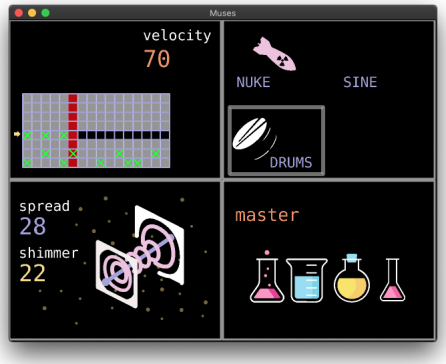


Figure (8) Screen short of Muses audio application

4 Interfaces in Practice

Until now we have looked at the uses of SVGs for DMI interfaces from a design perspective, presenting a framework for describing them. In this section we show the use of the framework in practice through the Muses audio application we have developed within the context of a larger research project, Printing the Muses, we are undertaking in conjunction with the Fine Print Research Centre at the University of West of England, into the design and implementation of Digital Musical Instruments. The Muses audio platform supports multiple sequencers, audio synthesis engines, effect sends, and a 4 track style digital tape recorder. For the most part it is written in the systems programming language Rust, supported by a selection of programs, e.g. the SVG interface compiler, written in the functional programming language Haskell. The architecture is modular in design supporting audio engines, for both synthesis and effects processing, to be added easily. An example screen shot of the application is given in Figure 8[13].

The audio and effect engines are developed in the DSP functional programming language Faust [16, 22], which is a small functional DSL capable of generating C++ and Rust, along with a selection of other backends. As standard in audio applications a high priority thread is utilized for audio, along with threads for handling the UI and input via Open Sound Control (OSC) [8][14].

The design and implementation of the Muses application is not the focus of this paper and instead the remainder of this section we cast our eye on SVG interfaces and their use for describing control surfaces for two customizable controller surfaces, the Sensel Morph and ROLI Lightpad.

[13]Muses 2D graphical interface was original inspired from the amazing audio Raspberry PI project OTTO, although we have diverged somewhat now. The is highlighted within the screen shot, where we have implemented a variant of OTTO's reverb UI.

[14]MIDI is also supported, but is intended for off the shelf interfaces that do not support OSC, e.g. keyboard controllers.

Attribute	Value
width	240mm
height	140mm
viewBox	0 0 230 130
interface_device	sensel

Table (2) Global SVG Attributes.

4.1 Printing Interfaces with Sensel Morph

The Sensel Morph[15] is a velocity sensitive, and swappable control interface that enables tactile response. It supports a pre-defined set of interfaces, allowing the user to define either MIDI or OSC message mappings, or the ability for custom interfaces via a C API. Sensel support users building custom interfaces for the Morph in two ways:

- The Sensel Application provides the ability to customize the messages produced, either OSC or MIDI, with off the shelf controllers, such as keyboard and drum pads, that Sensel sell. The application also supports defining simple, regular pads and sliders, in a simple GUI interface that can be printed out and configured for OSC and MIDI; and
- the Sensel API, which is a small C based API for communicating with the Sensel over USB, providing functionality for device discovery, configuration, and retrieving user contact information.

The interfaces presented in this paper are compiled to JSON and loaded via an application written in Rust, which using FFI calls to the Sensel API to process contact frames from the Morph via USB. The frames are processed and OSC messages are generated and communicated over UDP.

The Morph has a surface of interaction that is 230x130mm and the API can be configured to produce (x,y) and pressure for multiple touches against a high resolution sensor array made of a grid of 185x 105 of what Sensel terms "sensels". The device is configured such that when a contact is made, several sensels are activated, each having its own pressure reading, and the driver combines them to generate touch events. While it is possible to process the raw, CCD like date stream, directly, the Morph provides an embedded microprocessor that can analyze the image using computer vision techniques to identify individual touches, the force of each individual touch, the size, and so on. The maximum frame rate at which data can be read from the device is 2KHz.

Table 2 defines the set of "global" attributes for SVG interfaces. By requiring that the width, height, and view port map directly to the visible area on the Morph we avoid mapping and transform issues when performing tessellation and rasterization to the Morph JSON IR. Of course, we already preform basic clipping during rasterization and combining this with support for SVG transformations this restriction could likely be removed.

```
pub trait Controller {
    /// the name of this controller
    fn name(&self) -> &'static str;

    /// process a touch event, outputs
    /// OSC messages to socket
    fn touch(&mut self,
             contact: &Contact,
             socket: &UdpSocket)
            -> Result<(), &'static str>;
}
```

Figure (9) Controller Trait.

On start up the Morph backend loads the JSON IR creating a 2-dimensional non-mutable array representation of the IR buffer and then a 1D array of controllers is allocated, one element for each controller ID in the buffer, corresponding to the controllers field in the JSON IR. Elements of the controller array are instances of the Rust trait in Figure 9.

An implementation of this trait is provided for each controller type. Each instance, internally handles the maintenance of any state required to implement a controller's semantics, for example, a slider tracks the current position and outputs OSC messages of values within the range as specified in the JSON IR. The struct Contact contains data taken directly from the Sensel capturing information about the particular touch event.

For the most part individual controller implementations are straightforward. The endless controller, which allows for continuous rotation around a circle, is probably the mostly complicated implementation, as it tracks user movements around a unit circle, but is easily handled with basic trigonometry. Processing a touch event is reduced to just three memory reads, one for each of the look-up tables, and one for transferring control via a 'vtable' to the controller function itself. This is then followed by computing the OSC message, which in turn is written out (asynchronously) to the socket.

4.2 Controlling Grids with ROLI Blocks

ROLI's Lightpad Block[16] is a small wireless (Bluetooth LE) controller for musical expression. Like the Sensel it supports pressure based touch, but also enjoys a 15x15 matrix of bright RGB LEDS. Although a much smaller resolution Blocks can be "snapped" together with ROLI's DNA connectors, enabling building complex controller interfaces.

ROLI support users building custom interfaces for the Lightpad in a variety of ways, including:

[15] https://sensel.com/pages/the-sensel-morph

[16] https://roli.com/products/blocks/lightpad-m

Type	Attributes
pad	fill ="rgb(I,I,I)" preset=I
toggle	fill ="rgb(I,I,I)" preset=I
vertical slider	fill ="rgb(I,I,I)" preset=I
horizontal slider	fill ="rgb(I,I,I)" preset=I

Table (3) ROLI Lightpad Attributes.

- The ROLI Dashboard application provides a set of predefined interfaces for Lightpads, that can be customized with respect to the particular MIDI messages they produce;
- the Blocks code application supports compilation of Littlefoot programs, a small C like language for Block interface development, that can be uploaded and run on a Lightpad; and
- the Blocks SDK is a C++ framework for developing Windows, OS X, and Linux host applications that utilize Lightpad blocks, supporting compilation and installing of Littlefoot programs, direct communication, and more.

In general, the interfaces described in this paper are compiled to Littlefoot and the result can be used either with Blocks Code or the Blocks SDK to load interfaces on to a Lightpad. To date we have not utilized direct heap allocation and communication from a Lightpad to a driver application, but there is nothing in the approach described below that prevents such a development path.

Like the Sensel Morph the Lightpad is capable of recognizing multiple touch events, however, unlike the Morph it has a soft silicon surface, layered on top of a 15x15 matrix of programmable RGB LEDs. The reader might observe that x/y resolution of the Lightpad is considerably smaller than the Morph and places limitations on the kinds of controllers that can be easily represented, particular if more than a few are live at any given time.

To this end the set of possible controls is restricted, where we assume the attributes described previously, plus the additional ones shown in Table 3.

Specifying the **fill** attribute enables a control to utilize the Lightpad's tri-colour LEDs as can be seen from the example given in Figure 3. As **fill** is a standard SVG attribute the resulting SVG can be visualized as per any other SVG with the specified colour applied to the corresponding shape.

The **preset** attribute enables a control to be associated or grouped with a set of controls that are active only if the particular preset is active. For example, Figure 10 implements a Lightpad interface containing four pads, two, red pads, assigned to the preset zero and the other two, green pads, assigned to preset one:

This example highlights a downside to the approach for presets, as it means that displaying the resulting SVG in a drawing application, such as Adobe Illustrator, will present

```
pad !#x 0 !#y 0 !#size 3
    !#address "/midicc" # preset 0
    # iargs [0] # fill "rgb(255,0,0)"
<>
pad !#x 4 !#y 0 !#size 3
    !#address "/midicc" # preset 0
    # iargs [0] # fill "rgb(255,0,0)"
<>
pad !#x 0 !#y 0 !#size 3
    !#address "/midicc" # preset 1
    # iargs [0] # fill "rgb(0,255,0)"
<>
pad !#x 4 !#y 0 !#size 3
    !#address "/midicc" # preset 1
    # iargs [0] # fill "rgb(0,255,0)"
```

Figure (10) Preset example for ROLI's Lightpad.

one set of buttons on top of the other. On the other hand it demonstrates that a more programmatic approach to describing SVG interfaces can have further advantages.

Lightpads do not provide support for OSC and instead the generated Littlefoot produces and consumes only MIDI messages. This means that the receiving application must either handle MIDI messages directly or otherwise we provide a simple MIDI to OSC mapper application. The previously discussed Muses audio application, for example, supports both MIDI and OSC and has direct support for Lightpad interfaces that produce a set of predefined MIDI CC messages, plus MIDI note, pitch bend, and so on.

In general, generating Littlefoot is straightforward, the lack of any form of product or sum types requires that sets of global variables are allocated for all controllers with state, e.g. sliders, endless, etc., and functions are created that calculate properties for specific controllers, indexed via their IDs. Littlefoot expects callbacks to be implemented for touch events and these are additionally generated, mapping (x,y) touch to particular sliders. The process is somewhat complicated by the fact that Littlefoot does not support arrays and so the implementation converts the mapping function implied by the IR's buffer to a set of conditionals testing for a specific identifier. A draw callback is generated, which utilizes the required fill attribute to set the LEDs that map to a controller, as a function of the state of the controller, e.g. a section of slider will be black if the range is not covered by the current settings. Finally, presets are easily supported by adding an additional layer of indirection, again using conditionals as Littlefoot's only form of indirection, and the ROLI's control block or incoming midi messages are used to receive change requests.

Lightpad interfaces are a strict subset of the general SVG interface description described in Section 3 and the Sensel

Morph interfaces of Section 4.1, all be it with additional attributes for specifying fill colours and controller presets.

5 Discussion

In this paper we described a simple yet practical approach for the design and implementation of custom interfaces for musical expression. Its use was demonstrated with implementations for Sensel's Morph and ROLI's Blocks within the context of the design of new Digital Music Instruments, in this case the Muses application. While our design goals focus round the design and implementation of new DMIs the proposed system communicates over OSC or MIDI messages and as such can be used with many existing software and hardware digital musical instruments.

5.1 Related work

5.1.1 Musical Interfaces

The design and implementation of the interface for DMIs is an active field, often referred generally as New Interfaces for Musical Expression (NIME) in reference to the conference. Since its inception there has been a wide variety of work at NIME that looks theoretically at musical control of computers, in particular seminal work by Wessel and Wright that set out to report on the problems (or challenges) associated with the notion of the computer as a musical instrument [21]. Additionally Cook's early work also defined a number of principals for the design of computer musical instruments [5]. In the following decade authors such as Tanaka and Magnusson [13, 19] generalized these approaches to accommodate an ever expanding notion of computer, including the introduction of the smart phone. Today the market for control interfaces for musical expression is a huge multi-national operation, companies such as Ableton, originally a software only company, sell highly polished mass produced products in their thousands. The emergence of iOS and Apple's Audio Unit v3 has made the phone and tablet professional platforms for audio. However, with the exception of certain iOS applications the controllers have only limited features for customization, it is, of course, possible to control what messages they produce, but the interface's control surface is fixed during design and manufacture.

iOS applications such as Touch OSC[17] provide a modular approach to describing OSC control interfaces, however, unlike our approach they are limited and constrained by the iPhone and iPad screen interface. In particular, their interfaces provide no tactile control and their controls are the standard set of uniform shapes. Work by Glowacki [10], for example, combines the iPad's capacitive touch screen with a more tactile experience, with fabric overlays, however, to our knowledge this or a similar approach has not been applied in the area of interface design for musical instruments. It might be an interesting area of future work.

[17] https://itunes.apple.com/gb/app/touchosc/id288120394?mt=8

5.1.2 Output-Directed Programming

Direct manipulation interfaces, such as Adobe Illustrator, are useful in many domains, often providing interfaces accessible to designers and non-programmers. However, their lack of programmability in a high-level language makes it difficult to develop complex and reusable content. Output-Directed Programming, as described by Chugh el al [4, 11], aims to bridge the gap between direct manipulation and programming—their system Sketch-n-Sketch provides the ability to design and shape the desired output by example, e.g. dragging and stretching shapes on a canvas, and the application infers program transformations to match, possibly introducing variables, functions, and arithmetic relationships into the program. Conversely, the designer can add code programmatically and shapes are added to the canvas, which again can be directly modified with the mouse.

Sketch-n-Sketch lacks the domain specific focus of our interface DSL, rather looking at simple SVG generation and other similar applications, and is limited with respect to aspects of the program that can be directly synthesized. Originally Sketch-n-Sketch presented a simple Lisp like language to the design, but more recently replaced it with the web programming language Elm[18]. While Elm is a good fit for the browser based Sketch-n-Sketch it lacks some of the advanced features, in particular dependent types, utilized in the interface DSL. There seems to be a close synergy with respect to design goals between the two projects and it seems likely that ideas of Sketch-n-Sketch and the more general field of output-directed programming could provide a foundation for an interface design tool, merging features from both direct manipulation and programming in a high-level language.

5.1.3 Monoids and Diagrams

Monoids have gained particular attention recently in there use to model the denotational meaning of common computational structures, including Monads, Applicative Functors, and Arrows [18]. Our interface DSL is based on the mathematical structure Monoid and is an instance of the corresponding Haskell type class, enabling composition of controllers.

Monoids have also been used in the context of diagram specification, including SVG composition, and in particular the work on the Haskell DSL Diagrams [23, 24] was a key inspiration for our work and shares many ideas. In particular, Diagrams supports a compositional approach for describing images, similar to ours, however, the intention is different—Diagrams aims to provide a framework for programmatically generating images, while our goal is to enable both the design and implementation of musical interfaces.

[18] https://elm-lang.org/

5.2 Future work

Utilizing Adobe Illustrator to build users interfaces is on the one hand a practical solution as it is an industrial heavy weight when it comes to vector drawing, but comes with a heavy learning curve and features outside of the domain of interface design. In truth we utilize only a tiny subset of what Illustrator is capable of and in fact it is easy to build SVGs that are not compatible with our infrastructure, e.g. SVG gradients are not supported. More importantly defining custom SVG attributes is not well supported. To address this we plan to develop a simple SVG interface editor applying output-directed design combined with programming style editing of attributes and other interface properties, similar to the style of Hempel et al [11].

A feature often requested by users is the ability to map multiple output values, i.e. OSC messages, to a single controller, which in itself is straightforward, but of more interest and harder to achieve is the ability to morph from one set of values to another. It is likely that this would require some form of gesture recognition, via a Regression Neural Network (RNN), for example, similar to that described by Fiebrink and Francoise et al [6, 7].

An important area of future work is to consider the use of SVG interfaces to design and build non conventional controllers. For example, we are developing an instrument for studying and teaching polyrhythmic patterns, the Polyrhythmic Ring Sequencer, that utilizes SVG interfaces to describe a sequencer built on the Sensel Morph.

Finally an important area of future work in relation to our parent project, Printing the Muses, is to investigate the use of various approaches to printing tactile interfaces from SVGs, analyzing material properties and how they function under different real life conditions, e.g. in a night club, and how musicians handle differences in friction and other tangible qualities.

Acknowledgments

We thank our colleagues from the Computer Science Research Centre and Creative Technologies Lab for interesting discussions on all things programming, musical, and visual. In particular Tom Mitchell provided feedback throughout the development of SVG interfaces and the larger Muses project. Nat Roberton introduced us to Swell Paper and provided us with many enlightening discussions about design.

The OTTO audio project has been a huge inspiration for our work, particularly the Muses audio application. This work would not have been possible without Haskell's Diagrams package, whose design and ideas we have continually returned to for inspiration.

References

[1] Michael Abrash. 1997. *Michael Abrash's Graphics Programming Black Book, with CD: The Complete Works of Graphics Master, Michael Abrash* (10th ed.). Coriolis Group Books, Scottsdale, AZ, USA.

[2] Jim Blandy. 2015. *The Rust Programming Language: Fast, Safe, and Beautiful.* O'Reilly Media, Inc.

[3] Bert Bongers. 2000. Interaction Theory and Interfacing Techniques for Real-time Performance. *Trends in Gestural Control of Music* (2000), 41–70.

[4] Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. 2016. Programmatic and Direct Manipulation, Together at Last. In *PLDI'16: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 341–354.

[5] Perry Cook. 2001. Principles for Designing Computer Music Controllers. In *CHI'01: Workshop on New Interfaces for Musical Expression.*

[6] Rebecca Anne Fiebrink. 2011. *Real-time Human Interaction with Supervised Learning Algorithms for Music Composition and Performance.* Ph.D. Dissertation. Princeton.

[7] Jules Françoise. 2013. Gesture–Sound Mapping by Demonstration in Interactive Music Systems. In *MM '13: Proceedings of the 21st ACM international conference on Multimedia.* 1051–1054.

[8] Adrian Freed and Andy Schmeder. 2009. Features and Future of Open Sound Control version 1.1 for NIME. In *NIME '09: Proceedings of the Conference on New Interfaces for Musical Expression.* 1–5.

[9] Jean Gallier. 1999. *Curves and Surfaces in Geometric Modeling: Theory and Algorithms.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[10] B.R. Glowacki. 2018. Mixed play spaces: Augmenting digital storytelling with tactile objects. *Interactions* 25, 2 (2018), 58–63.

[11] Brian Hempel and Ravi Chugh. 2016. Semi-Automated SVG Programming via Direct Manipulation. In *UIST '16: Proceedings of the 29th Annual Symposium on User Interface Software and Technology.* 379–390.

[12] Sergi Jordá. 2005. *Digital Lutherie Crafting musical computers for new musics' performance and improvisation.* Ph.D. Dissertation.

[13] Thor Magnusson. 2010. Designing Constraints: Composing and Performing with Digital Musical Systems. *Computer Music Journal* 34, 4 (2010), 62–73.

[14] Simon Marlow. 2010. Haskell 2010 Language Report.

[15] Eduardo Reck Miranda and Marcelo Wanderley. 2006. *New Digital Musical Instruments: Control And Interaction Beyond the Keyboard (Computer Music and Digital Audio Series).* A-R Editions, Inc., Madison, WI, USA.

[16] Y Orlarey, D Fober, and S Letz. 2004. Syntactical and Semantical Aspects of Faust. *Soft Computing* 8, 9 (9 2004), 623–632.

[17] Juan Pineda. 2005. A parallel algorithm for polygon rasterization. In *SIGGRAPH '88: Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, Vol. 22. 17–20.

[18] Exequiel Rivas and Mauro Jaskelioff. 2017. Notions of Computation as Monoids. *Journal of Functional Programming* 27 (2017).

[19] Atau Tanaka. 2010. Mapping Out Instruments , Affordances , and Mobiles. In *NIME '10: Proceedings of the Conference on New Interfaces for Musical Expression.* 15–18.

[20] W3C. 2011. Scalable Vector Graphics (SVG).

[21] David Wessel and Matthew Wright. 2002. Problems and Prospects for Intimate Musical Control of Computers. *Computer Music Journal* 26, 3 (2002), 11–22.

[22] Dominique Fober Romain Michon Yann Orlarey, Stéphane Letz. 2017. FAUST Tutorial for Functional Programmers. In *FARM 2017: Proceedings of the 5th ACM SIGPLAN International Workshop on Functional Art, Music, Modelling and Design.*

[23] Ryan Yates and Brent A. Yorgey. 2015. Diagrams: a Functional EDSL for Vector Graphics. In *FARM 2015: Proceedings of the 3rd ACM SIGPLAN International Workshop on Functional Art, Music, Modelling and Design.*

[24] Brent A Yorgey. 2012. Monoids: Theme and Variations (Functional Pearl). In *Haskell '12: Proceedings of the 2012 Haskell Symposium.*