o.io: a Unified Communications Framework for Music, Intermedia and Cloud Interaction

Adrian Freed, David DeFilippo, Rama Gottfried, John MacCallum, Jeff Lubow, Derek Razo, David Wessel

CNMAT
University of California Berkeley
1750 Arch Street
Berkeley, CA 94709
wessel@cnmat.berkeley.edu

ABSTRACT

We present work on the "o.io" system, a suite of tools for hiding vendor-specific and protocol- specific details of controllers and actuators and for replacing diffuse documentation and heterogeneous ontologies with harmonized, situational schema carried along in real-time with gesture and actuator control values as Open Sound Control (OSC) bundles. We introduce useful general design patterns and object-oriented tools that support them. We conclude with details of support of two particular devices that illustrate the potential of "o.io", the QuNeo and Bluetooth LE heart rate monitors.

Keywords

Cyber-physical Systems, Open Sound Control, OSC, Device Integration, *Lingua Franca*, Discovery Protocol, Software Engineering

1. INTRODUCTION

The rapid evolution of sensor and actuator technologies is facilitating the development of a plethora of new gesture sensing devices and interfaces for music and media applications. Legacy devices - game controllers, MIDI devices, biometric sensors, lighting controllers - abound as well. Could there be a harmonizing framework for dynamic mashups of these devices with a unified encoding and programming model? This question is setting the agenda for important aspects of the research on the "internet of things" and cyber-physical applications like robotics.

Computer music, with its concerns for low-latency and temporally coordinated interaction among a variety of control and audio devices, for fault tolerant behavior, for rapid deployment in a performance setting, and for agile and rapid prototyping, is now a model discipline for such a research agenda.

We report on our recent steps to provide such a harmonizing framework. We have had access to CNMAT's large and diverse collection of devices. We rely on Open Sound Control (OSC) the widely used encoding that has served as a *lingua franca* for interactive music and media projects. And we exploit the o-dot (o.) programming extensions we have made to the Max and PD environments.

Along the way we demonstrate some essential software engineering practices for the development of music and media systems that can exploit a collection of past, present, and future devices.

1.1 Related Work

Examples of existing applications and frameworks used to harmonize communication among devices include: the now-defunct GlovePIE for Windows, the actively-supported OSCulator [7], TUIO [5] and VRPN[10]. GlovePIE and OSCulator support a small number of devices and the developers don't provide an API for third parties to add new devices. VRPN, on the other hand, is not proprietary and has drivers for the relatively large number of devices favored in the niche application space of VR scene navigation and control. TUIO provides a simple OSC namespace and the dynamic semantics for modeling surface interactions with object collections.

Our ambition is to create an open, scalable framework for diverse devices within and between niche application spaces. We have explored over thirty different devices so far. Scaling to thousands of different kinds of devices in dozens of application spaces will require extraordinary consensus, cooperation and effort, the bulk of which will be done outside our small development group. Our initial focus, described herein, is therefore exploration of efficient, agile development techniques–a prerequisite for broad adoption.

1.2 Vision and Contribution

Our main contribution consists of hardware and software tools (called "o.io") that support the requisite design patterns include transcoding, interpreting, state caching, logging, helping, simulating, and bridging. We use these tools to unify communication among a wide variety of devices chosen to explore the most challenging dimensions of the project. We routinely use "o.io" in teaching, music, and intermedia production projects. Our vision is that users will move from having devices at hand to building applications in minutes rather than hours or days. A typical concrete scenario is to type "o.io." into a fresh Max/MSP or PD object box followed by the model name of your device (e.g. "leapmotion"). When you plug the device into USB, the "o.io.leapmotion" object outputs a stream of OSC messages whose interpretation is self-evident without needing to consult manuals or a standard's documentation.

This dream is related to the old story of "plug and play" often lampooned as "plug and pray". We favor stateless models without registries, complex discovery protocols, API's, query schemas or static ontologies. Our "o.io" functions implement a simpler model that might be called "plug and spray" because we output a continuous stream of OSC bundles that describe a sequence of "snapshots" of the entire state of the device.

1.3 Paper Structure

We first describe the architecture of "o.io" from the "bottom up" moving from the concrete encodings of bits to higher layers that interpret these bits. We offer a more specific, illustrated account of our support for a touch surface device, the QuNeo and contrast this with how "o.io" is used with Bluetooth LE devices supporting the protocol's rich service discovery mechanisms. We conclude with a summary of the thorny problems encountered requiring substantial future work and outside collaboration.

2. ARCITECTURAL CHOICES

2.1 Introduction

"o.io" is a suite of tools that hide vendor-specific and protocol-specific details of controllers and actuators and replace diffuse documentation and heterogeneous ontologies with harmonized, situational schema that are carried along in real-time with the gesture and actuator control values as OSC bundles.

2.2 OSC

Aspects of OSC favorable for this project include:
Stability: the specification has not substantially changed since it was first introduced and there are no signs of immanent change. OSC has been resilient to outside influence by any particular corporate or government interests.
Efficiency: OSC uses binary data supporting commonly used primitive data types.
Exercised: OSC is widely used in diverse communities and has support in all the major programming languages and application frameworks. Many applications and data streams can be easily bridged to.
Adoption: Wide adoption would suggest that it is relatively easy to understand and work with.
Expressive: OSC supports the major primitive data types, hierarchy, and time stamps.
Lightweight: As simply an encoding, OSC doesn't carry the baggage of complex protocols.

OSC does have disadvantages for our purpose: names are not Unicode so names privilege languages with Latin alphabets; time tags are not yet widely implemented and no organization is actively enforcing OSC as a standard–an activity that could promote interoperability.

2.3 The "o." platform

We chose the "o." platform [4] to build "o.io" with because it has native support for OSC, it affords modern, agile programming techniques, and it has recently been ported to PD, and Node Red, and is therefore accessible for free with generous open source licensing.

The "o." system was designed to bolt onto dataflow languages such as Max/MSP and PD endowing them with robust support for aggregate data structures. The computational heavy lifting in "o." applications is the "o.expr" component, a rich expression language supporting vector arithmetic, type polymorphism, functional programming, symbolic processing, higher order functions, and functions as first class values.

Although "o." is a relatively small C library it is not easy to port to microcontrollers with small memory footprints or without floating point support. In these cases, we use the smaller, simpler C++ class library: "OSC for embedded control and Arduino." [3]. A further limitation of "o." at this time is that it has not been ported to other media programming environments, web platforms or tablet devices. Alternatives for this research we considered include Javascript, TCL/TK and Lua. "o." has the performance advantage over many languages of not using garbage collected storage. Performance is an important concern as we scale data rates while maintaining tight latency constraints.

2.4 Narrative Description of "o."

The following description of "o.io" is narrated in the general direction of raw device data flowing from devices to become rich, cooked OSC bundles. The same narrative in reverse tells the story for control and actuation applications.

There is a small but important asymmetry to mention with these two directions of flow: protocol implementations of UDP, for example, usually provide a return address for response packets. These addresses can be bound to incoming sensor packet data and carried through to the final cooked OSC data so that applications can send information back to the data source.

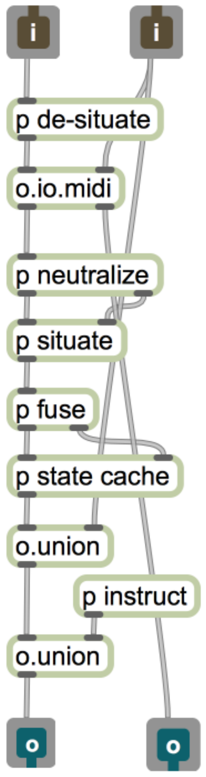


Figure 1. "o.io.quneo" Main Patch

3. IT'S A WRAPPER

The examples that follow use the code abstraction method of the host language (nested patches) and the data abstraction method of "o." (delegation [9]). We call the style of object-oriented programming that combines the two abstraction methods "wrapping" because each layer, in an onion-skin nesting of patches, selects components of incoming OSC packets, processes relevant data, and delegates the rest up to surrounding layers.

Figure 1 is the outermost wrapping patch for the o.io.quneo device. In the language of class-based object-oriented programming we would say that "o.io.quneo" subclasses "o.io.midi". We avoid this terminology as it does not do justice to the dynamic nature of the data flows in "o.io"

3.1 OSC and Embedded OSC

To maximize reuse and minimize development effort, we have found it fruitful to transcode from raw device data to OSC as early as we can in the processing stream. For sensor module devices that produce analog signals or sensors with digital serial protocols such as SPI and I2C, we use the "OSC library for embedded microcontrollers and Arduino" with a Teensy 3.1 ARM-based microcontroller [1]. This OSC library extends the Arduino stream class so it can be used to exploit existing Arduino protocol libraries including TCP/IP, USB serial and asynchronous hardware UART serial.

A representative example of this approach is "o.io.esplora" for the Arduino Esplora gamepad/sensor platform. This device has buttons, joysticks, an LDR, microphone, and accelerometer for inputs and an RGB LED for output. OSC bundles are communicated as slip-encoded USB serial byte streams. The o.io.esplora module has been exercised for two semesters now in a lab that supports an introductory class on music programming.

Another device with embedded OSC encoding is x-OSC from x-io [2], a platform that communicates OSC payloads wirelessly as UDP packets. This is our primary rapid development tool for music and media projects.

3.2 Transcoders

We have implemented transcoders for the following common protocols: USB HID, USB Serial, MIDI, UDP, and Bluetooth LE. Transcoders do the minimum of interpretation of the raw data stream from each protocol. For example, if it is known in advance that a particular UDP port will carry OSC encoded packets (e.g. x-OSC) then "o.io.udp.osc" can be used which does no transcoding. Otherwise "o.io.udp" is used which simply places each byte of a UDP packet into an OSC BLOB array. For USB HID, "o.io.hid" transcodes parameter id/value pairs into OSC bundles. "o.io.midi" leverages baseline MIDI parsing support by transcoding into OSC bundles containing MIDI messages.

3.3 Packetizers

Some data sources and sinks require packetization and depacketization. TCP and USB serial are common examples where we provide "o.io.slip" to buffer streams of bytes and identify packet boundaries.

3.4 Interpreters

The "o.io" modules described so far are protocol-centric rather than device-centric and accordingly they constitute a growing, reusable core. Interpreting o.io modules, on the other hand, are device specific and are composed from protocol specific handlers and basic "o." functions used to name, normalize, and structure parameters. Now that we have built a solid core of protocol support and associated helper functions, we find it is these interpretive functions that require the bulk of the development time assigned to each device.

3.4.1 Motivations and usability

Before describing this workflow in detail we introduce some motivating principles and goals for the interpreting modules.

The primary goal of interpreting device data as OSC messages is to improve the usability of systems built with the devices. What frustrates usability is the great diffusion of information about the parameters being represented. Some of it is in device user manuals, some is in operating system configuration panels, some is in special configuration modes of the device itself and much is implicitly referenced in protocol standard's documents. For many devices useful information is deliberately obscured by obfuscation and encryption protocols, further diffused in documents that resume efforts to reverse engineer and recover these trade secrets. What users and device integrators need is dynamic, valid information at hand as the device is integrated into a system and used.

3.4.2 Situators

A good number of devices we work with have labeled controls. These labels are clearly a better basis for a parameter namespace than the numerical encodings typical of the MIDI, and HID standards, for example. We call the process of replacing meaningless numbers with meaningful names "situation" because we are reorganizing the data to support the situation of the user rather than the situation of the developer or standard's document authors [8].

Many devices have no labels or will be repurposed so the labels are not meaningful. These new situations require development of a custom namespace. The namespace design problem is rich and interesting enough in itself to be the subject of a future paper. Here we summarize a few practical observations on namespace design.

3.4.2.1 Multiple Interpretations

The x-OSC module is representative of devices in related ecosystems (e.g., Arduino) that are specifically designed to be customized and incorporated into larger systems. Many music and media projects employ such platforms. Each of x-OSC's 32 I/O pins can be set to different electrical interfacing conventions: analog, digital, PWM etc. These pins will be connected to sensors and actuators of the integrator's choosing. The integrator is best situated to name and interpret the data acquired and sent to the pins. In a way directly analogous to extending a class in a traditional object-oriented software system, we provide a baseline "o.io.xosc" module that the user extends by re-writing its stream of OSC bundles.

The underlying "o." system has a cloning semantics that afford the addition of data to OSC bundles as they pass through processing steps. This promotes the accretion of multiple interpretations and reinterpretations of raw in-coming device data.

Part of the namespace design problem is deciding when the benefits of multiple interpretations outweigh the extra cognitive burden of studying larger packets.

3.4.2.2 Normativity

When concerned that a namespace might be legible to a very narrow community of users, we search for a functionally equivalent namespace that might be more generally accessible. For example, we use chessboard notation (letter/numeral) to identify elements in two-dimensional grids. This avoids many ambiguities inherent with row/column, column/row, bottom to top, and top to bottom conventions.

3.4.3 Neutralizers

The numerical domain of parameter values varies from device to device. Often the domain represents an implementation bias such as the values resulting from an A/D convertor or constraints in the number of bits available in a byte or nibble. We find it fruitful to scale parameters into the unit interval [0,1] using IEEE 32-bit floating point numbers. For devices with a center detent or spring return such as joysticks we will use the interval [-1,1]. The intention is to be "value neutral" favoring neither designers or users and to support rescaling by simple multiplications.

This unit scaling is useful when, for example, updating a device from 10-bit A/D conversion to 12-bits. This change is seamless when applications are built using the unit interval and the device itself does this "neutralizing" from "hardware" units.

3.4.4 Calibrators

Calibration processes both scale and label parameter values according to an established norm. We favor standards that are broadly used, well documented and with stability of consensus and dissensus, e.g. SI units. Where calibration requires the storage of device specific scaling parameters, we note that the user experience improves greatly if the devices themselves can store parameters in non-volatile memory. Examples of this include x-OSC and the Nintendo Wiimote.

3.4.5 Fusers

Fusion involves the temporal and spatial aspects of parameter streams. Spatial grouping is conceptually straight forward and proceeds using the idea that proximate values are grouped in the OSC bundle representation. Grouping mechanisms include: coordinates in OSC lists, similar parameters in the same OSC sub-bundle, and hierarchical design of the OSC namespace. Groupings can involve rich topologies involving connectedness by spatial position, function, color, interaction modality and physical constraints (e.g. joystick ordinates).

Temporal fusion can be very challenging–especially for older protocols, where timing is an implicit feature of the transport protocol, e.g. MIDI and RS232. OSC bundles were originally invented to enable representation of concurrently sampled data or describing the desired state of a system at a future time. The QuNeo is an example where a single gesture (pressure applied to a silicone pad surface) results in a sequence of MIDI messages represented without explicit framing. By experimentation it is possible to determine the order in which MIDI sequences are issued and thereby provide reliable packetization and state representation.

3.4.6 Unifyers

Devices such as keyboards, mice, music keyboards, and gamepads exist in numerous minor variants of a core theme. With unification we harmonize the parameter values and address space. Almost all gamepads, for example, are composed from the same elements: switches, joystick, and a button array. Studying a dozen USB HID gamepads, we found that the numbering of switches differed between devices and operating systems–as did the range of joystick values. Scaling to the unit interval ("neutralization") suffices for unification of joystick values. The switches require a custom mapping to a namespace unified among joysticks. The success of unification is measured by whether different models and devices can be substituted for each other in a particular application.

Our most ambitious experiments whith this unification design pattern are associated with organizing support for motion capture devices such as the Kinect affording an "o.io.skeleton.*" suite to serve gesture processing and mapping applications in a device-independent manner.

3.4.7 Validators and Simulators

For complex or weakly documented devices it can be very challenging to develop and validate "o.io" modules. We have learned to be suspicious of diffuse documentation and even the ontological data that protocols like USB are required to provide in the device implementation. What matters to the user is what the device they have in hand actually does not what a potentially outdated document says. This problem is exacerbated when devices, their drivers or API's are field upgradeable.

Building a simulation of the device itself is a good way to address these difficulties. These simulations serve as proxies when the physical device is unavailable in addition to aiding in the verification of the implementations. Often we can put the physical device and its on-screen simulation side by side and operate each independently. This approach is strongly related to the model/view/controller design pattern.

We have found the effort to build simulations pays off in terms of time saved testing and validating and discovering what devices actually do interacting with them. Simulations also have a pedagogical value as they afford rapid sketches of potential future variations of a device.

To be clear, we are not proposing completely faithful emulations of a device; too much is lost gesturally in many cases anyway when mouse or multi-touch interactions substitute for device interactions. Our focus with these simulations is to expressively manifest the gestures in a visual mode that complements and is parallel to the lexical representation in OSC.

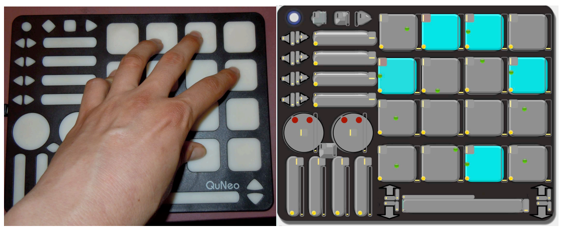


Figure 2. QuNeo and its Simulation

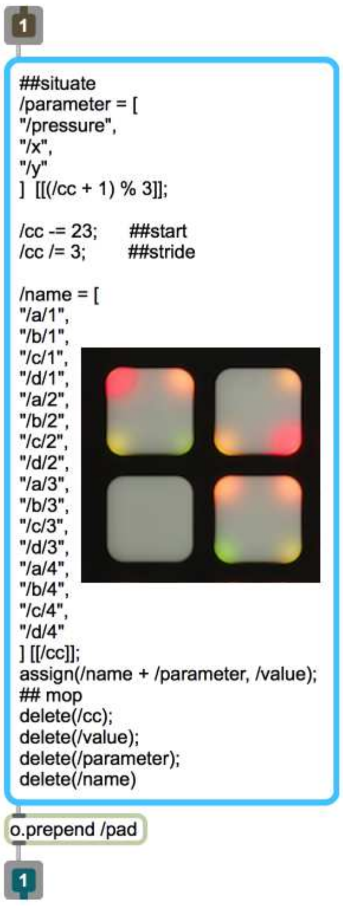
3.4.8 State cachers

When computer data rates were slow compared to processing rates, it became the habit of IO device designers to minimize the bit representations of data and use a "send on change" approach to schedule communications. We have noticed that applications can be enormously simplified with an alternative, stateless-protocol approach where no assumption is made that a receiver has an accurate representation of the state of a system. This requires that the entire state of the device be transmitted when a change is to be communicated. For the many legacy devices that don't do this we implement a cache that models and reflects the state of the device.

We have developed helper functions for common device semantics such as buttons. A single bit sufficient to represent that state of a button is represented in o.io OSC streams using four messages, two boolean values that represent the state of the switch (up/down) and two that represent transitions (pressed/released). We also provide an OSC time stamp and sequence number to detect lost packets. Bundles there represent valid and complete representations of a past device state and mechanism to detect lost packets. These are prerequisites for the subtle details that have to be implemented well to support communications in high data loss wireless environments, a problem carefully managed in the RTP-MIDI standard [6].

4. CASE STUDY: o.io.quneo

The QuNeo is a small, flat control surface with a heterogeneous array of silicone position and pressure sensors back illuminated with colored LEDs. (See Figure 2). It is interfaced using USB with MIDI encoding. Interactions with each sensor produce pairs of values represented as MIDI "note on/off" and "control change" messages. There is no simple, natural mapping from the keyboard/controller orientation of MIDI to the spatial arrangement of the QuNeo buttons, sliders and pads. This is reflected in a decision tree (built with "o.cond") that is required to demultiplex the streams according to which sensor group messages refer to. Data from each path in the decision tree trickle down to patches like the ones shown Figure 3 where data from the two MIDI message types are combined into a single, human-readable OSC message using simple modulo arithmetic operations and table lookups.



```
##situate
/parameter = [
"/pressure",
"/x",
"/y"
] [[(/cc + 1) % 3]];

/cc -= 23;     ##start
/cc /= 3;      ##stride

/name = [
"/a/1",
"/b/1",
"/c/1",
"/d/1",
"/a/2",
"/b/2",
"/c/2",
"/d/2",
"/a/3",
"/b/3",
"/c/3",
"/d/3",
"/a/4",
"/b/4",
"/c/4",
"/d/4"
] [[/cc]];
assign(/name + /parameter, /value);
## mop
delete(/cc);
delete(/value);
delete(/parameter);
delete(/name)
```

Figure 3

Spatial location of the 16 control pads of the QuNeo are named according to a chessboard grid from the viewpoint of White, where the bottom leftmost pad is /a/1 and the top rightmost pad is /d/4. This example illustrates unusual features of the o.expr language: provision for dynamic operations on names and a functional incarnation of the assignment operator.

The named (situated) gesture streams flow to a common collection point to be prepared for output with "o.collectwithtimeout." This collects the incoming encoded data using a 3ms lease to aggregate 'concurrent events,' – i.e. touching multiple pads at once. An "o.union" object caches the state of the QuNeo merging state changes into the stored bundle as required. When the lease expires and there has been a change of state, the entire OSC bundle stored by "o.union" is dispatched.

5. CASE STUDY: o.io.bluetoothle

There are no unified mechanisms in any operating system for discovery of what devices are attached to a system and what services are available from them. Each operating system offers protocol specific APIs. We generally address this difficulty with a separation of concerns reflected in the second inlet of the "o.io" functions presented so far. This second inlet is used to define which particular device an "o.io" function handles the stream of. Separate "o.io" functions handle enumeration and implement user interfaces for the selection of particular devices.

BluetoothLE is a relatively new protocol we have added support for that has an interesting approach to enumeration that may be the solid basis of a more general enumeration scheme. We outline a particular example we have used in intermedia, dance and music projects that use wireless BluetoothLE heart rate sensors for each dancer.

When a BluetoothLE heart rate monitor (HRM) begins transmitting data, the "o.io.bluetoothle" object receives a callback from the operating system's Bluetooth API with information about the peripheral. In response, the object transcodes the data into an OSC bundle and outputs it. If the ambient patch appends the message [/discover/services true] and feeds it back to "o.io.bluetoothle" the object calls the BluetoothLE API to ask for enumeration of the services the device provides. In response to the OS callback for each service, o.io.bluetoothle sends an OSC bundle containing information about the service into the ambient patch. This time around the ambient patch can append the message [/discover/characteristics true], whence a service-characteristic request will be issued and the resulting enumeration of the characteristics will be transcoded and output into the patch. Characteristics are the fine-grained attributes that can contain single value, usually a sensor measurand. These values are obtained by appending one of the following messages to the characteristic bundle and sending it up to o.io.bluetoothle [/read true], [/notify true], or [/write = <value>].

This call and response between "o.io.bluetoothle" and the host environment is implemented as a reentrant coroutine to avoid being interrupted by the Bluetooth LE API. It also allows for a mostly stateless implementation of "o.io.bluetoothle". We find this delegation style of programming [9] easier to understand, and more reliable than traditional threads or state machine implementations.

An important challenge in general purpose wrapper design is making something flexible enough to support functionality not anticipated at the time the wrapper was built. "o.io.bluetoothle.hrm" was tested with two brands of HRMs. To support features that future hardware may provide, OSC bundles containing information about unrecognized services and characteristics are delegated to the ambient patch via the right-most outlet. This allows a user to extend the functionality of "o.io.bluetoothle.hrm" via another layer of wrapping.

6. CHALLENGES AND NEW WORK

From the usability standpoint we have identified a major difficulty with many legacy devices: they don't have unique identifiers even if their underlying protocol supports unique IDs or serial numbers. It is really common in interactive music and media projects to combine several devices, one for control with each hand for example. Without unique device ID's consistent enumeration to identify which device is on the left or right is impossible.

We will continue to encourage new device developers to take the extra step of including a unique ID. For legacy devices we are developing shim hardware to insert unique ID's in the data stream. This will be an opportunity to also add time tagging, transcode early in the protocol stream and avoid the problem with USB keyboards, mice and trackpads that the host operating system monopolizes their data streams wrapping in them a proprietary API.

In conclusion, we have demonstrated the viability of "o." for a wide variety of devices sufficient in number to be confident of our workflow and design patterns. We look forward to addressing further the questions of scaling by distributing the system and supporting external developers.

Acknowledgements
This work was supported in part by the TerraSwarm Research Center, one of six centers supported by the STARnet phase of the Focus Center Research Program (FCRP) a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

7. REFERENCES

[1] Teensy 3 ARM Microcontroller. 2014. http://www.pjrc.com/store/teensy3.html.

[2] x-OSC Wireless Controller. 2014. http://www.x-io.co.uk/products/x-osc/.

[3] CNMAT OSC for Embedded Control and Arduino. 2013. https://github.com/CNMAT/OSC.

[4] Freed, A.M., J.; Schmeder, A. Composability for Musical Gesture Signal Processing using new OSC-based Object and Functional Programming Extensions to Max/MSP *NIME 2011*, Oslo, 2011.

[5] Kaltenbrunner, M., Bovermann, T., Bencina, R. and Costanza, E. TUIO: A protocol for table-top tangible user interfaces *6th International Workshop on Gesture in Human-Computer Interaction and Simulation*, 2005.

[6] Lazzaro, J. and Wawrzynek, J. An Implementation Guide for RTP MIDI, RFC 4696, November, 2006.

[7] Osculator Osculator. 2014. http://www.osculator.net.

[8] Suchman, L.A. *Plans and situated actions : the problem of human-machine communication*. Cambridge University Press, Cambridge [Cambridgeshire]; New York, 1987.

[9] Taivalsaari, A. Delegation versus concatenation or cloning is inheritance too. *SIGPLAN OOPS Mess.*, *6* (3). 20-49, 1995.

[10] Taylor II, R.M., Hudson, T.C., Seeger, A., Weber, H., Juliano, J. and Helser, A.T., VRPN: a device-independent, network-transparent VR peripheral system. in *Proceedings of the ACM symposium on Virtual reality software and technology*, (2001), ACM, 55-61.