URMUS – AN ENVIRONMENT FOR MOBILE INSTRUMENT DESIGN AND PERFORMANCE

Georg Essl

University of Michigan
EECS & Music, Ann Arbor, U.S.A.
gessl@eecs.umich.edu

ABSTRACT

UrMus is a multi-layered environment intended to support interface design, interaction design, interactive music performance and live patching on multi-touch mobile devices. Its design is based on suggestions from the HCI community for design for multiple target audiences and design for creativity. In addition UrMus is intended to be an environment for use directly on mobile devices and was not designed with legacy assumptions taken over from desktop computing. UrMus offers multiple component which all are replacable and offer ways to control computational cost, easy of programming and design as well as separation of representation and function. Ultimately UrMus tries to stay as flexible and independent from any particular design choice as possible and rather is an environment where mobile interaction design becomes possible.

1. INTRODUCTION

UrMus is an environment to support interface design, interactive media performance and mobile music making on multi-touch mobile smart phones. The design of UrMus is driven by multiple goals, but also by a certain set of assumptions about mobile computing and needs for future mobile interactions. The field has recently seen a drastic acceleration, yet there is still a certain lack of infrastructure to allow to broadly explore what mobile music making can mean.

Mobile phones have become attractive platforms for audio and multimedia processing. However, the approach to using them for this purpose either is based on developing special purpose software that will offer only one solution, or is based on ports of existing audio and multi-media solutions to mobile phone platforms. Our goal is to offer a generic multi-media processing environment very much in the spirit of Max/MSP, pd, SuperCollider or ChucK. We believe that mobile devices are not just small computers, but have inherently different I/O capabilities. For example input modalities are, in a traditional computing paradigms treated as external, exemplified by the short-hand HID (for Human-Interface-Device). Mobile devices themselves are, however themselves the sum of all input and output capabilities. A

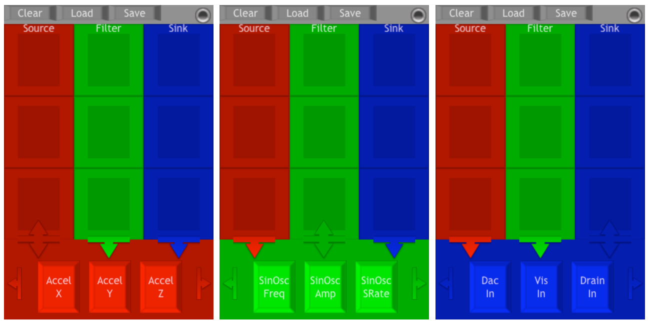


Figure 1. The default UrMus interface ready to select sensor sources (left), filter or other manipulation flowboxes (center), or actuator sinks (right).

second concern is that the prevailing editing paradigms for laptop computers assume the standard interface hardware of full alpha-numeric keyboard, a mouse and a large display.

Hence we look to start from scratch and develop a multimedia processing environment that is directly designed for the platform for which it is intended. This includes diminishing or removing the role of alpha-numeric input, replacing single-point mouse interaction with multi-touch interactions, designing for small, often partially occluded screen display, taking motion sensing such as accelometer and magnetic field sensing into account from the start. A similar goal was already explored in a project called SpeedDial [6] which however is intended for smart phones with 12-key input. We have since developed an environment that looks to be broader and specifically addresses multi-touch interactions. We call this project UrMus and the current paper describes the overall design of the environment.

2. RELATED WORK

UrMus builds upon many lessons learned from SpeedDial [6], a mobile phone synthesis environment for SymbianOS smart phones, yet is a completely new design. There are numerous project that address interaction design on for mobile devices. Probably the closest to the current project is RjDj, a commercial environment using pure data as the audio engine[1]. One personal motivation to develop UrMus is academic. The author was looking for an open environment to use for research and teaching, and found that the current landscape of mobile environment are often too tightly mixed with commercial interests to allow open exploration. At the same time we see this as an opportunity to incorporate a broad range of design options into a new setup.

Audio processing engines have a long-standing history going back to its origins with Music I by Max Matthews. Ultimately multiple paradigms have emerged to support the process of generating computer music. The most dominant paradigms are text-based systems, such as CSound [1], Arctic/Nyquist [5, 3, 2], SuperCollider [10] or ChucK [20] on the one hand, and graphical patching systems, such as Max/MSP [13] or pure data (pd) [11, 12] on the other hand. For a more detailed review of audio processing languages and systems see [19].

UrMus tries to not commit to any particular paradigm per se, but rather looks to offer an environment in which many different paradigms can be instantiated. It features the support for full 2D UI design. In a less general form this can also be seen in RjDj as well as other iPhone apps such as MrMr[2]. MrMr main design is that of an Open Sound Control remote client with configurable UI based on predefined widgets. UrMus bases its UI design on more general concepts such as regions and widgets can be derived and designed from those. Also UrMus does not look to be an OSC remote, but is primarily intended to incorporate all processing, whether audio, or multi-media directly on the mobile device.

The idea of having performance interfaces has previously appeared in various forms. So does Max/MSP through jitter and other means offer ways to create interactions. In fact the Max/MSP and pure data interfaces themselves mix in performance elements through widgets. The Audicle serves a similar purpose for ChucK [21]. The idea of Audicle to offer multiple representations, called faces, is very much related to UrMus. Some aspects of the Audicle are rather closely tied to the needs of live coding and the underlying language ChucK. UrMus is in principle not tied to such concerns but is designed open enough to be useful in this fashion.

Vessel is a multi-media scripting system based on Lua [18]. In this sense it is closely related to UrMus. However UrMus' goals are rather different from Vessel's. The primary function of Lua in UrMus is not to serve to script multi-media and synthesis functionality but rather to serve as a programmatic API and a middle layer between lower level functionality. For example the synthesis computations in UrMus' data flow engine UrSound are fully realized in C, whereas Vessel is designed for algorithmic generation. However with increased computational performance on mobile devices one could see merging the ideas of Vessel and UrMus and this might be facilitated by the fact that they already share the same scripting language. Serpent [4] is a Python-based embeddable scripting solution which emphasized functional programming. Lua is in many ways similar to both Serpent and Python. We see Lua's main advantages in the ease of embedding, its run-time performance and the simplicity of the syntax.

3. DESIGN GOALS

The primary design goal for UrMus is to be a generic editing environment for multi-touch mobile smart phones such as the iPhone. We consider this to ultimately be a completely new HCI setup and many decisions that have been made for previous systems designed for desktops have to be reevaluated. Design goals are:

- Direct and inherent support of all accessible input modalities through sensor technology as available on the device.

- Direct and inherent support of all accessible output modalities through actuator technology as available on the device.

- The environment should largely be *neutral* to assumptions of best interaction paradigm, i.e. it should allow many concurrent or competing interaction paradigms. Ultimately this means the environment is conceptualized as a meta-environment that then allows to create a wide range, if possibly any, desirable interaction paradigm.

- The environment should be multi-layered and accessible at all levels [14]. A multi-layered environment has multiple advantages. On the interaction side, it allows for design of simple interfaces that still at a lower level can be manipulated to offer increasing complexity. The interface can be accessed at the level of expertise of the user. On the performance side, it allows to implement solutions at the layer appropriate to achieve the required computational speed for the task.

- Offer *design-by-molding*. Refactoring and redesigning from scratch is time-consuming and does not allow exploration. The environment should be designed around the idea of modifiability to allow the user to explore variations without having to frequently rewrite data structures or interface designs. This is supported by another goal:

- Clean separation and "pluggable" interfacing [15, 17]. If one has a synthesis algorithm that allows to play

[1]`http://rjdj.me/`
[2]`http://poly.share.dj/projects/#mrmr`

chromatic notes, the choice of layouting of a multi-touch interface should not interfere with this algorithm. This requires a clean yet flexible interfacing mechanism between synthesis engine and UI. If one can simply replace one UI with another and have it operate seamlessly with an existing synthesis setup we call this property ”pluggable”.

- Allow for On-the-fly patching. This means multi-media interfaces and media streams should be repatchable on-the-fly with minimal or no concern to the user.

In terms of functions UrMus wants to offer the following:

- Design of multi-touch user interface and interactions. Ideally on and for the phone at the same time.
- Efficient multi-media dataflow pipeline.

4. THE DESIGN

UrMus has a three-layered design. The bottom layer are core engines that are written in system-close languages. Most parts of UrMus are written in C, with some exceptions being made for C++ to accomodate existing open software such as STK, and Objective-C to interface with Apple's system libraries.

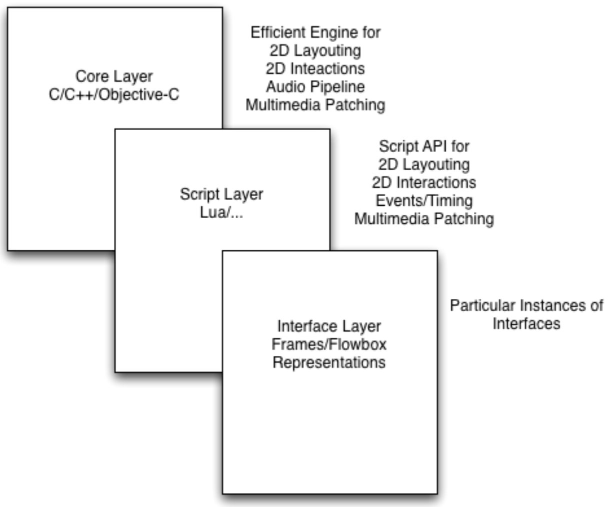


Figure 2. The three layers of UrMus.

The second layer is a higher level script language which offers abstraction of the primary design functions of UrMus. We chose the script language Lua [9] over other options for numerous reasons. Most important is the malleability that the language offers which is compatible with our design-by-molding design goal. Lua offers strong typing of primary data types, yet weak typing of data structures. This design goal means that data structures can very easily be extended or modified without refactoring. Lua also offers functions as first order types. This allows functions also to be modified without refactoring through a mechanism called hooking. Secondarily Lua is a very compact and efficient script language hence also attractive from a performance perspective. Lua comes with a very generous license. Didactically Lua is attractive because it is widely used in the gaming industry hence draws a lot of interest from students who seek to enter this industry.

The Lua layer has to offer accessible abstraction for the main functionality of UrMus, which is multi-touch interfaces and interactions as well as multi-media data flow. UrMus offers these through distinct object APIs. The interface API is inspired in part by the addon interface API of the popular game World of Warcraft. We found that aspects of this API satisfy many of the needs for our purpose and gives students an attractive point of access. However the WoW API does not support multi-touch, accelerometers, and other mobile-specific interaction modalities and hence the UrMus interface API in many ways different to this API. The data flow API is designed from scratch to support a novel data flow engine based on normed generic patching. We call this system UrSound and it is described in more detail separately [7].

An important component of the Lua layer are events. Events inform about occurrences and can be used to define interactions. Any part of the UI can register for events. When registered, whenever new data is available from a sensor such as an accelerometer, a designated callback function in Lua will be called. That way UI elements can be written to respond to event based actions.

The third layer are particular instances of Lua programs in UrMus that offer a complete running program or UI interface. Currently UrMus offers a default mapping interface that also goes by the name UrMus that allows to map out data flows using multi-touch. It is important to note that this interface is by no means canonical. Many other interfaces for mapping can be implemented in the UrMus environment. The default interface serves as a first working example and as a prove of concept. At the same time it implements a certain type of visual mapping paradigm that is somewhat different than the wire-patching paradigm of Max/MSP or pure data on the one hand and text based patching such as via SuperCollider or ChucK on the other hand.

5. CORE LOW LEVEL ENGINES

The core low level engines currently consist of the following entities: 2D layouting and events, media dataflow pipeline, Lua runtime-compiler and run-time engine. This is extensible and we anticipate a detailed networking engine, and a 3D engine to be added.

All these engines are accessible from the next higher

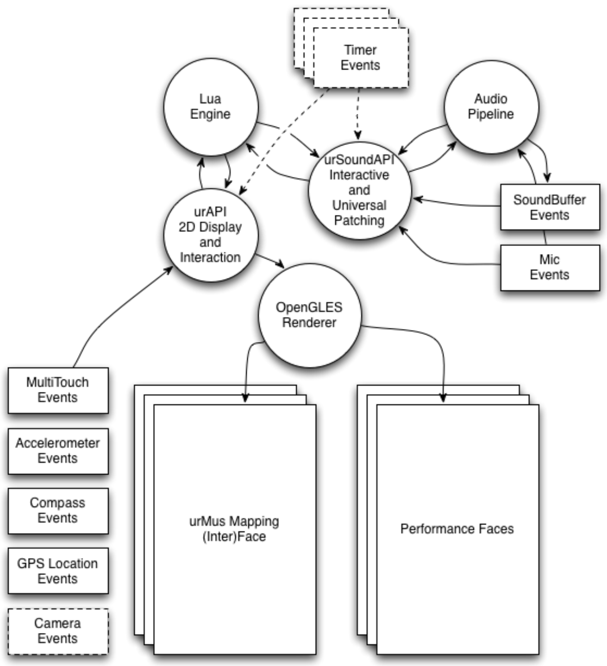


Figure 3. Interrelation between different components of UrMus.

layer and are, on this lowest layer for the most part independent. The reason for this is that we actually want core parts to both operate independently and be replaceable. For example if one wants to use the UrMus environment with a pd or ChucK audio-pipeline, this should be possible either as a replacement for the current pipeline or as an addition to it. To allow this it is necessary to ensure sufficient separation by design.

5.1. 2-D layouting and interactions

The 2-D layouting engine is responsible for offering all aspects of two-dimensional layouting. It is a layouting engine written from scratch in OpenGLES hence promises to port to all platforms supporting that standard without relying on higher level libraries.

The basic paradigm borrows from the notion of a *region* that can be found in the interface API of World of Warcraft. Regions serve as the basic UI element for everything visual as well as the size for accepting events coming from interactions. One can think of a region as any arbitrary rectangular region associated with the screen. A region may or may not be visible. In fact regions come with ways to control their own visibility. However even invisible regions do serve an important function in that they can be used to define regions of interactions that do not require extra visual representations. Regions can be manipulated in size and position and can be associated with colors. The main visual display of information within regions is achieved through textures. By associating rectangular regions with textures one can build complex looking visual interfaces. Regions support relative anchored layouting. One can anchor relative to another region and hence build groups of regions. If the parent region in the group is moved or otherwise has its layout changed, all anchored children will have their layouts refreshed. This way it becomes possible to have complex user interface elements that can be dynamically modified and still behave seamlessly even if there are complex layouting relationships. Anchoring can also help order elements and facilitate insertions and deletions.

Many of these aspects can already be found in interface layouting systems of computer games such as World of Warcraf, however the urMus engine extends these existing systems in numerous ways. For example textures can be drawn into via 2-D drawing primitives. This is roughly modeled after the programming environment processing. But as any region can have an attached texture and any texture can be used as brush in painting, one can implement a range of visual metaphors such as visual recursions or interrelations between layouts and painting.

Each region can be informed of a range of events. In order to receive an event, a region registers a function to be called when the event occurs. Events are typically associated with some form of user interaction. An event can occur when a touch event starts within a region, when a finger is dragged into a region, when a finger is lifted, or when a finger is dragged outside the region. There are also events for dragging and resizing the region, for scrolling the region vertically or horizontally, and for double tapping. All these are events associated with multi-touch interactions. There are further events that trigger from other forms of input such as accelerometer data, compass data and so forth. The role of this layer is to expose all this functionality in a way that can be integrated by a higher level language layer.

5.2. UrSound: The multi-media Pipeline

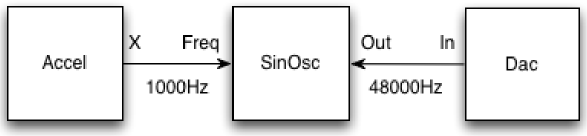
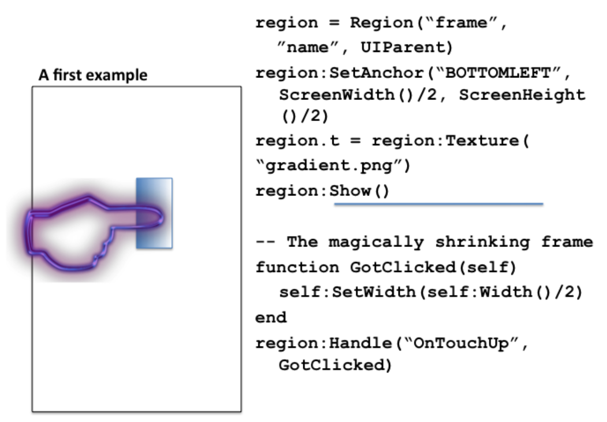


Figure 4. A simple example data flow in UrSound. Accelerometer X axis data pushes into the frequency parameter of the sine oscillator. The dac pulls from the sine oscillator output. Each happens at its independent rate and each connection uses normed semantic-free data.

The multimedia pipeline is written to incorporate all sensors and actuators into a multi-rate data flow. Here a connected network flow from sensors through filter and other manipulation algorithms and synthesis engines to actuators such as loud speakers, visual display, and vibrotactile display can be established.

It is important to note that the multi-media pipeline in some sense duplicates functionality with the 2D layouting engine. Both can take sensor input and both have ways to relate these inputs to visual outcomes. This duplication is a result of the design goal of keeping each engine independent. One can either choose to use them in an interface setting or in a data flow setting. A mixing of the two paradigms can happen, but only at a higher level. This ensures that each engine is autonomous. The UrSound engine is written at the lowest level to avoid any performance loss due to intermittent script language processing. The engine has a number of properties that are design criterion for UrMus such as basic support of on-the-fly mapping through concepts such as normed data flow and multi-rate processing (see Figure 4 for an illustration). Details of the pipeline are beyond the scope of this overview and can be found in a separate publication [7].



```
region = Region("frame",
  "name", UIParent)
region:SetAnchor("BOTTOMLEFT",
  ScreenWidth()/2, ScreenHeight
  ()/2)
region.t = region:Texture(
"gradient.png")
region:Show()

-- The magically shrinking frame
function GotClicked(self)
  self:SetWidth(self:Width()/2)
end
region:Handle("OnTouchUp",
  GotClicked)
```

Figure 5. A full example of creating an interactive region in UrMus.

5.3. Lua engine

UrMus uses the Lua programming language. Its run-time compiler and run-time engine are freely available and easily integrate in a C environment. In UrMus it serves as a unified scripting environment with added API for higher level access to the low level engines. We use an unaltered version of lua-5.1.4 stripped of libraries incompatible with the iPhone SDK.

The Lua engine then allows to expose the functionality of the other core engines in Lua through added global library functions, user data types or methods associated with these user types.

6. LUA AND THE URMUS LUA API

Lua is a scripting language with a reputation of being fast and light-weight. In fact cross-language benchmark studies have indeed shown that Lua performs very efficiently for a run-time script language [22]. It is also very attractive for use in project because it is designed to be easily embeddable.

While having simple syntax, Lua offers very attractive and powerful data structures through dynamic associative arrays called tables that serve as the only organizing mechanism within the language. Every other instance is a first order type. Lua's syntax is a hybrid of C-like, Pascal-like and some C++/Java like object-oriented notation. Overall Lua's syntax looks mostly procedural though in reality Lua can be used as an object-oriented as well as functional programming language. The standard reference for the language is [9].

The UrMus Lua API adds global functions, global variables as well as functions to create user data which itself comes with methods. To take two examples:

```
local r = Region()
r:SetHeight(200)
r:SetWidth(120)
```

This creates a user data called Region, and uses two methods of regions to set height and width. Regions can be informed of events. This can be achieved by setting callback functions for each event type:

```
function WiggleRegion(self,x,y,z)
      self:SetAnchor(x*320,y*480)
end

r:Handle("OnAccelerate", WiggleRegion)
```

This will make the region move along the screen according to the values of the accelerometer x and y axis data, scaled to match the screen dimensions. Full documentation of the UrMus Lua API is distributed with UrMus.

The UrSound dataflow engine can also be accessed through a Lua API. Similar to regions, flowboxes, the data processing units of UrSound are instantiated explicitly. Each provided flowbox of UrSound has a global prototype that can be used as a factory to create new instances. The function to create a new instance is `FlowBox(type, name, prototype)`. For example

```
mySinOsc = FlowBox("object",
        "mySinOsc", _G["FBSinOsc"])
```

creates a new instance of the SinOsc flowbox from the global prototype FBSinOsc. Some flowboxes are state-less and hence do not require instancing. Many hardware sources and sinks are of this type. For example we can use

```
dac = _G["FBDac"]
```

to get the global instance of the audio playback sink. In order to establish a pull link between the sin oscillator and the dac we use the method outflowbox:SetPullLink(inindex, inflowbox, outindex).

```
dac:SetPullLink(0, mySinOsc, 0)
```

The moment this is executed, a sine will play at default frequency of 440 Hertz and at normed amplitude. To connect this to an accelerometer that pushes into the frequency we do the following:

```
accel = _G["FBAccel"]
accel:SetPushLink(0, mySinOsc, 0)
```

Details of the technical mechanisms that govern UrSound can be found in [7].

The UrMus Lua API allows for constructing of a very broad range of user interfaces and interactions. The goal here is two-fold. One is to offer generic editing environments. However there are a range of paradigms how to do this and UrMus does not commit to any one paradigm, rather any paradigm can in principle be implemented using the Lua API. The second is to offer a way to define performance interfaces for specific musical instruments or multi-media interactions implemented in UrMus. That is if one wants to design a specific specialized interaction, this should be possible in UrMus as well.

7. INTERFACES

7.1. Default UrMus Mapping Interface

The default UrMus mapping interface serves as a proof of concept of the interface and interaction possibilities of the UrMus Lua API. It demonstrates complex interactions such as multi-touch dragging, multi-touch scrolling, dynamic layouting and more. At the same time it also implements a specific form of graphical patching that creates links implicitly from location. While similar in name this is not the same thing as implicit patching [16]. Full details of the UrMus default interface will be published elsewhere [8]. For the current discussion we will briefly discuss its main idea.

The core idea of UrMus is the support of rapid establish and change of signal flow through an interface. Hence the process of connecting or disconnecting should be as simple and quick as possible. In order to achieve this, a fixed grid-like arrangement of rows is created where flow elements can be moved. If two elements are horizontal neighbors of each other they will automatically connect. If they are moved out of place the connection is automatically broken. Hence we get a form of implied patching by position. In comparison to graphical line patching paradigms such as Max/MSP or pd this removes the step of creating the line between unit generators.

The visual space is limited so the UrMus interface contains numerous mechanisms to help manage space. By double-tapping between rows and columns new elements can be inserted and if the size of the patch exceeds the visual view, multi-touch horizontal and vertical scrolling helps navigate.

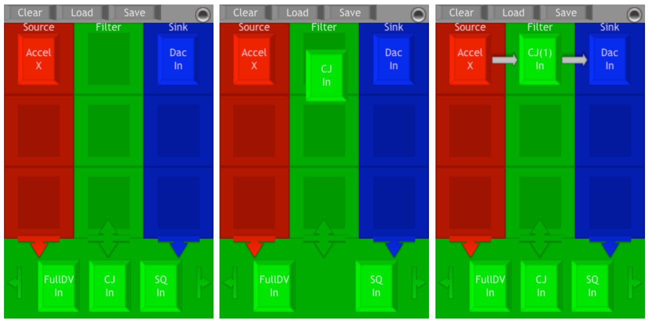


Figure 6. The default UrMus interface for on-the-fly patching.

7.2. Example Instrument Faces

Numerous instrument interfaces have already been designed using UrMus. Figure 7 shows a few examples. The top row are interfaces designed by students at the University of Michigan during a class that used UrMus as environment for mobile phone instrument design.

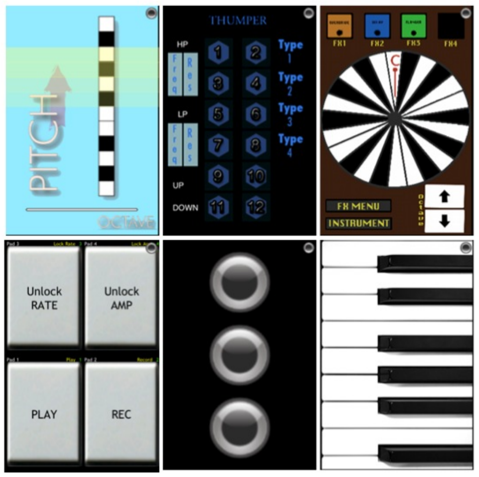


Figure 7. Six examples of instrument interfaces. Top left was designed by Rishi Daftuar and Raphael Szymanski, top center by Colin Zyskowski, top right by Colin Neville and Owen Campbell.

Overall these interfaces are highly interactive and can change their appearance and mapping as needed, yet can be realized with short scripts. These interfaces take a few dozen lines of Lua code to realize. The most ambitious interface to date is the default UrMus interface which consists of just over two-thousand lines of Lua source code. A powerful aspect of UrMus is the separation of engine from representation. For example a large range of different patches can be used by any of the interfaces shown in Figure 7 which were designed for pitches. As long as the semantics of the interaction regions is specified in some standard fashion, the interfaces become simply replaceable. For example the user can then simply decide whether to prefer a flute or a piano interface for performance, without changing the sound. Or vice versa one can keep an interface and replace the underlying sound synthesis method without difficulty.

8. CONCLUSIONS

UrMus is a meta-environment for the design of multi-media interactions and is specifically designed to allow general development of mobile phone musical instruments. It offers separation of representation from processing engines and hence allows flexible modular design and a choice of preferred interaction by both the user and the designer. UrMus is still in its infancy and we hope to extend it further, by strengthening its networking capabilities, keep adding support for more sensors and providing more rendering engines, specifically 3D rendering and interactions. UrMus is available at:

`http://urmus.eecs.umich.edu/`

9. ACKNOWLEDGEMENTS

Thanks to Alexander Müller who helped design the UrMus default interface and provided numerous other interface suggestions. Also thanks to all students in the class "Building a Mobile Phone Ensemble" and patiently handling UrMus in early, immature forms of existence, and for designing exciting musical instruments in UrMus. Thanks also to Roger Dannenberg for stimulating discussions on sound mapping and on Serpent.

10. REFERENCES

[1] R. Boulanger, *The Csound book: perspectives in software synthesis, sound design, signal processing, and programming*. Cambridge, MA, USA: MIT Press, 2000.

[2] R. Dannenberg, "The implementation of nyquist, a sound synthesis language," *Computer Music Journal*, vol. 21, no. 3, pp. 71–82, Fall 1997.

[3] ——, "Machine Tongues XIX: Nyquist, a Language for Composition and Sound Synthesis," *Computer Music Journal*, vol. 21, no. 3, pp. 50–60, Fall 1997.

[4] ——, "A Language for Interactive Audio Applications," in *Proceedings of the International Computer Music Conference (ICMC)*, 2002.

[5] R. Dannenberg, P. McAvinney, and D. Rubine, "Arctic: A Functional Approach to Real-Time Control," *Computer Music Journal*, vol. 10, no. 4, pp. 67–78, Winter 1986.

[6] G. Essl, "SpeedDial: Rapid and On-The-Fly Mapping of Mobile Phone Instruments," in *Proceedings of the International Conference on New Interfaces for Musical Expression*, Pittsburgh, June 4-6 2009.

[7] ——, "UrSound – live patching of audio and multimedia using a multi-rate normed single-stream data-flow engine," 2010, submitted to the International Computer Music Conference.

[8] G. Essl and A. Müller, "Designing dynamic mobile music instrument interfaces with UrMus," 2010, in preparation.

[9] R. Ierusalimschy, *Programming in Lua, Second Edition*. Lua.Org, 2006.

[10] J. McCartney, "Rethinking the computer music language: Supercollider," *Comput. Music J.*, vol. 26, no. 4, pp. 61–68, 2002.

[11] M. Puckette, "Pure data: another integrated computer music environment," in *in Proceedings, International Computer Music Conference*, 1996, pp. 37–41.

[12] ——, "Pure data: Recent progress," in *Proceedings of the Third Intercollege Computer Music Festival*, 1997, pp. 1–4.

[13] ——, "Max at seventeen," *Comput. Music J.*, vol. 26, no. 4, pp. 31–43, 2002.

[14] B. Shneiderman, "Promoting universal usability with multi-layer interface design," in *CUU '03: Proceedings of the 2003 conference on Universal usability*. New York, NY, USA: ACM, 2003, pp. 1–8.

[15] C. Simone, M. Divitini, and K. Schmidt, "A notation for malleable and interoperable coordination mechanisms for cscw systems," in *COCS '95: Proceedings of conference on Organizational computing systems*. New York, NY, USA: ACM, 1995, pp. 44–54.

[16] L. F. Teixeira, L. G. Martins, M. Lagrange, and G. Tzanetakis, "Marsyasx: multimedia dataflow processing with implicit patching," in *ACM Multimedia*, 2008, pp. 873–876.

[17] N. Villar and H. Gellersen, "A malleable control structure for softwired user interfaces," in *TEI '07: Proceedings of the 1st international conference on Tangible and embedded interaction*. New York, NY, USA: ACM, 2007, pp. 49–56.

[18] G. Wakefield and W. Smith, “Using lua for multimedia composition,” in *Proceedings of the International Computer Music Conference*. San Francisco: International Computer Music Association, 2007, pp. 1–4.

[19] G. Wang, “A History of Programming and Music,” in *Cambridge Companion to Electronic Music*, N. Collins and J. DEscrivan, Eds. Cambridge University Press, 2008.

[20] G. Wang and P. R. Cook, “Chuck: a programming language for on-the-fly, real-time audio synthesis and multimedia,” in *ACM Multimedia*, 2004, pp. 812–815.

[21] G. Wang, A. Misra, and P. R. Cook, “Building collaborative graphical interfaces in the audicle,” in *NIME ’06: Proceedings of the 2006 conference on New interfaces for musical expression*. Paris, France, France: IRCAM — Centre Pompidou, 2006, pp. 49–52.

[22] E. Wrenholt, “Fractal benchmark,” 2007, retrieved 6/1/2010 at http://www.timestretch.com/FractalBenchmark.html.