Designing Mobile Musical Instruments and Environments with urMus

Georg Essl
Electrical Engineering & Computer Science
and Music
University of Michigan
gessl@eecs.umich.edu

Alexander Müller
Deutsche Telekom Laboratories
TU-Berlin
a.mueller@telekom.de

ABSTRACT

We discuss how the environment urMus was designed to allow creation of mobile musical instruments on multi-touch smartphones. The design of a mobile musical instrument consists of connecting sensory capabilities to output modalities through various means of processing. We describe how the default mapping interface was designed which allows to set up such a pipeline and how visual and interactive multi-touch UIs for musical instruments can be designed within the system.

Keywords

Mobile music making, meta-environment, design, mapping, user interface

1. INTRODUCTION

UrMus is a meta-environment written to allow the flexible design of sound and media synthesis systems, as well as to support the design of mobile music instruments. We have a long history of systems intended support music generation and sound synthesis with computers. However these were designed for computers with an interaction paradigm that is different to that of a mobile device. The standard interaction modality for desktop and laptop are keyboard, mouse and a sensibly large screen. In mobile devices these have been replaced by multi-touch input, dial-keys, accelerometers, and a screen size that is limited by the size of a pocket. However, things are not all grim for the mobile device. Certain interactions, such as hand gestures are more natural for a mobile device due to the ergonomic relationship of the form factor to our motor abilities. Hence we felt that it warrants starting anew in designing a sound synthesis environment that allows to design interactions that more closely match the abilities that are possible on mobile devices.

The goal of this paper is to show how urMus can be used to design mobile music instruments. We also explain and describe the default mapping interface. However this paper will not go into detail about the architecture of urMus itself. Those details can be found in separate publications [7,8]. Here we will focus on the use of urMus for user interface design.

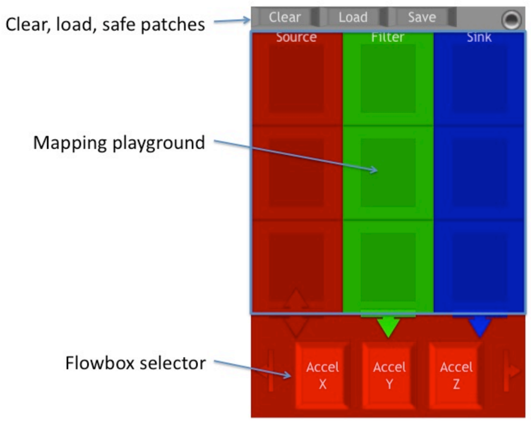


Figure 1 - The default urMus mapping interface.

2. Overview of urMus

UrMus is a meta-environment. It is not currently meant to propose one solution to interaction design or interface for a synthesis engine on a multi-touch mobile device. Rather it offers a way to create a wide range of such interfaces and hence allows the exploration and comparisons of various prototype proposals. The environment ultimately serves the goal of supporting the development of interactions on and for mobile devices. It currently consists of two engines: one for 2-D layouting and one for multi-media dataflow. Both are accessible through a higher level scripting language Lua [10]. This means that fully flexible UI design that is not bound to a pre-defined set of UI widgets is fully integrated with data flow that links sensory input through data processing to actuator output. The goal is to keep the building blocks as general as possible so that a wide range, perhaps most conceivable mobile phone interactions can be realized within this environment.

2.1 Related Work

A primary associated goal in the design of urMus is to keep it inherently non-paradigmatic. This means that it should not espouse or implement one particular interface or interaction paradigm but rather serve as an environment where many, perhaps all can be implemented. Eaglestone and co-workers suggested that there might be multiple cognitive styles that dictate how composers interact with synthesis software and that one should design systems with these cognitive styles in mind [5]. This work very much attempts to follow this program, by not ab initio fixing one particular paradigm. Even more so the boundaries between paradigms are continuously being blurred.

So do existing text-based synthesis environments often over ways to create other forms of interaction. For example the Audicle serves as a visual support, interaction and display system for the ChucK programming language [20] and Maui is a GUI design layer within ChucK [21] both allowing graphical elements to be introduced in an otherwise heavily text based paradigm. Nyquist has also been expanded to offer multiple forms of representations recently [3]. ixiQuark [11] plays a similar role with respect to SuperCollider [12]. At the opposite end, script based objects have been introduced for Max/MSP and pd [14,15,16]. For example a lua~ object exist for Max/MSP [21] as well as a chuck~ for pd [9].

The initial and a driving motivation behind designing urMus is the creation of an environment that is distinctly suitable for the mobile platform. It should supports mobile interaction modalities natively and naturally. In fact this motivation was the original reason to start this project. In this context SpeedDial is a direct precursor [6] based on 12-key smartphones.

There are projects 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 [17]. It offers offline interface design and uses a pd engine for sound rendering. MrMr is an OSC remote control system with text-configurable UI based on predefined widgets [13]. Vessel is a multi-media scripting system based on Lua [21]. 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 and high-level interactions. For example the synthesis computations in UrMus' data flow engine UrSound are fully realized in C, whereas Vessel is designed for algorithmic generation. UrMus is designed on principles of multi-layered design [18], and design for variation [19,20].

3. DESIGNING MOBILE MUSICAL INSTRUMENTS

Designing a mobile musical instrument can be viewed as the process of taking input from device sensors, transforming them through some algorithmic means into output which is then displayed through some of the device's actuators which can be speakers, display or any other modality. This is a broad prescription and can have a wide range of realizations. For example a mobile music instrument may simple link the signal received from a built-in accelerometer to the frequency of a sine oscillator, which then is played back through the speaker. Or it may be a complex sequencer that allows polyphonic music to be written by interacting with a dense set of visual elements on the multi-touch screen.

Contemporary multi-touch-based mobile smart-phones have a range of sensory input capabilities. By far the most complex of these sensors is the multi-touch input, not only because it allows for simultaneous channels that additionally is directly coupled to visual display. UrMus offers a detailed multi-touch UI engine that is inspired by the UI design offered in popular computer games. In addition it provides a multi-rate dataflow pipeline, which connects unit generators and algorithms into a dataflow.

Through urMus's Lua API both these engines can be made to interact. To understand how this works in urMus we discuss both parts in turn, beginning with the dataflow engine.

3.1 Setting up data flows

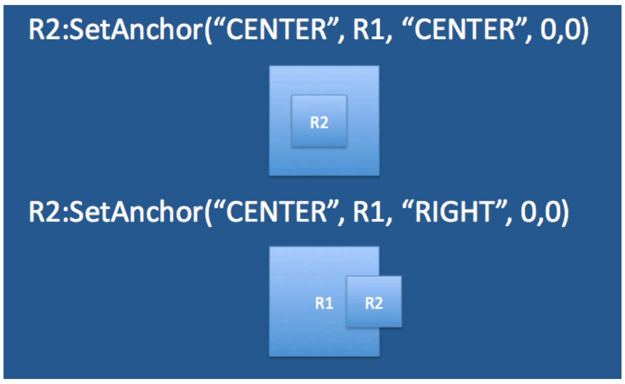

R2:SetAnchor("CENTER", R1, "CENTER", 0,0)
R2
R2:SetAnchor("CENTER", R1, "RIGHT", 0,0)
R1
R2

Figure 2: Effect of anchoring regions.

UrMus's dataflow engine urSound resembles existing synthesis engines in many ways. Connection between elementary processing blocks are established which then prescribed in which order a time sequence of data is to be processed. UrSound does not have an inherent master data rate, nor does it define any canonical control rate. Rather the dataflow can be on multiple rates and the rate of connected components (or if needed external timers) defines the local rate of the data flow network. In most cases this is not something that an author has to be aware of as it often is natural to let each component operate at a rate that is offered by its connected components. However a consequence of allowing natural rates to dictate the local rate is that data flow can go both upstream and downstream, which specifies at which end of the flow a rate may be specified. For example the accelerometer data is updated at a native rate of 1000Hz. If the data flow is upstream (or "pushed" in urSound parlance) then a processing block connected to the accelerometer will be fed at the rate of the accelerometer. At the other end of the pipeline, if the data stream flows downstream (or is "pulled") the flow on the output side will dictate the rate. For example if a unit generator is pulled from the audio-output pipeline, the natural rate of the audio-pipeline will propagate to the unit generator. Furthermore inputs and outputs are signed normed signals. That is, they always have a range of [-1, 1]. This means that processing blocks can always be connected functionally without having to specify scale transformations. The semantic of the inputs and outputs is implied in the processing block. Details on this can be found in a separate publication [8].

For our purpose here the most important mechanism is the way interactions can be fed into the dataflow pipeline. The pipeline allows numbers to be "pushed" or "pulled" programmatically. This can be used to feed user interactions on a graphical user interface into the pipeline. In this case the data rate is based on the user's actions.

To illustrate all this let us explore two examples, for which we will use the pseudo-notation `->` for a push link and `<-` for pull link.

```
Accel(x)->SinOsc(freq)<-dac
```

This data flow pushes accelerometer data into the frequency of a sine oscillator and the dac pulls samples from the sine oscillator.

```
Push(1)->SinOsc(freq)<-dac
Push(2)->SinOsc(amp)
```

This data flow has two separate push instances that drive input to the sine oscillator frequency and amplitude (sharing the same instance). This instance of the sine oscillator is pulled by the dac. The push object is similar to both the number input and the bang object in graphical patch languages such as pd, except that both functions have been merged.

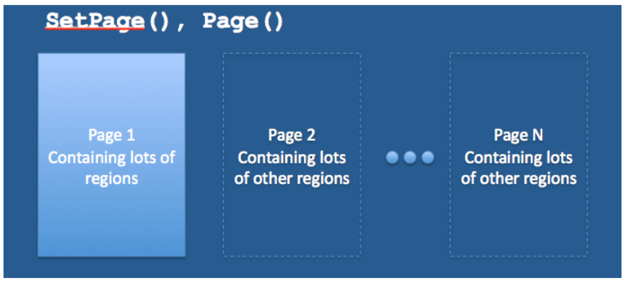

SetPage(), Page()
Page 1 Containing lots of regions
Page 2 Containing lots of other regions
Page N Containing lots of other regions

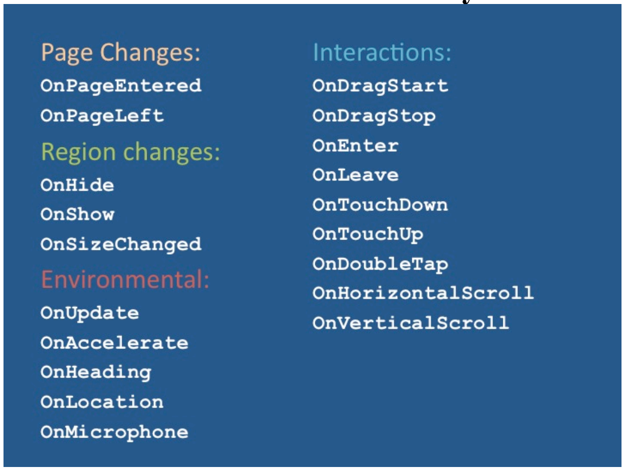
Figure 3: Pages allow for a visual name space mechanism for regions.

3.2 Core Layouting Functionality

Anchors are a main mechanism for layouting. This concept also exists in other UI layouting systems, such as Apple's Interface Builder and is modeled closely after the API provided by World of Warcraft. Rather than specify an absolute position, all layouting happens because of a relative position to another region. Figure 2 shows an example of a change in layout due to a change in anchoring. Anchored regions inherit many layouting properties of their parents. For example if a parent is moved, all regions anchored to it will move also, visibility rules do propagate to children, making it easy to hide complex grouped regions while treating as a single entity. Anchoring also makes it easy to do ordering operations such as insertions of regions between two adjacent ones by treating the anchor relationship like a linked list.

In order to allow separate visual pages that are easy to manage, there is paging mechanism, which essentially serves as a visual name-space (see Figure 3). New regions are always associated with the currently active page and regions are only visible if the page is active to which they have been associated. However the Lua name space itself remains global. Hence multiple pages can easily share functionality or dataflows.

3.3 Core Interaction Functionality



Page Changes:
OnPageEntered
OnPageLeft

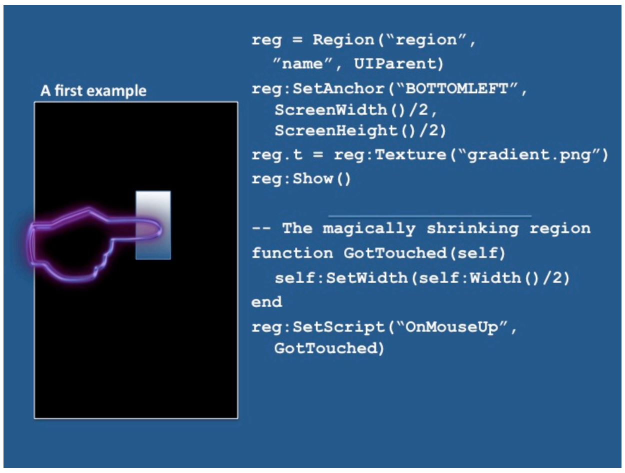
Region changes:
OnHide
OnShow
OnSizeChanged

Environmental:
OnUpdate
OnAccelerate
OnHeading
OnLocation
OnMicrophone

Interactions:
OnDragStart
OnDragStop
OnEnter
OnLeave
OnTouchDown
OnTouchUp
OnDoubleTap
OnHorizontalScroll
OnVerticalScroll

Figure 4: Events supported by urMus.

UrMus uses events to propagate information that is not part of the standard program flow. There are essentially two broad types of events: Those triggered by some kind of user generated action or sensory input, and those generated by UI related changes. Only regions can ever be informed by events and events are inherently associated with its region. For example there are a range of touch-related events, which trigger if the event happened to this region. For example OnEnter will trigger if a moving touch event enters a region, and OnLeave will trigger when it leaves again. OnDoubleTap triggers if the region is double-tapped. OnUpdate informs a region that the current UI layout is about to be rendered in OpenGLES and hence allows frame-rate-dependent adjustments to be written. Events for all supported sensory input are available and offer each region to independently react to it. For example one can easily write a UI that consists of many regions which each randomly and in different ways react to accelerometer input because each will be handling this event separately. The list of all currently available events is shown in Figure 4.



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

-- The magically shrinking region
function GotTouched(self)
  self:SetWidth(self:Width()/2)
end
reg:SetScript("OnMouseUp",
  GotTouched)
```

Figure 5: Example of a region taking and responding to an input event.

To see this in action consider the code example depicted in Figure 5. It creates a region, textures it and sets it up for a very simple UI interaction that will successively half the region width with each touch event.

3.4 Writing Interactive Interfaces

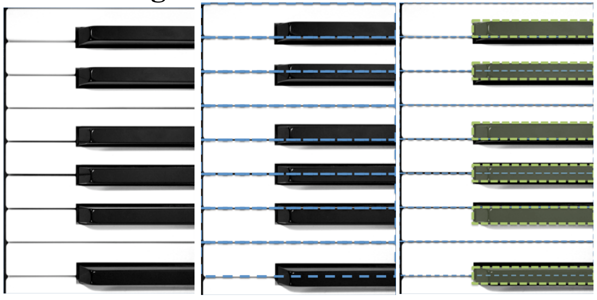


Figure 6: Example of a region taking and responding to an input event.

To see how one can design a fitting interface that will interoperate with a dataflow consider the example of creating a very simple piano keyboard. First we find a free image of an octave of a piano keyboard. Then we "instrument" the image by placing regions over the image keys. A rectangular region covers each white key. We do the same for the black keys. These regions serve two functions. One is to take input and

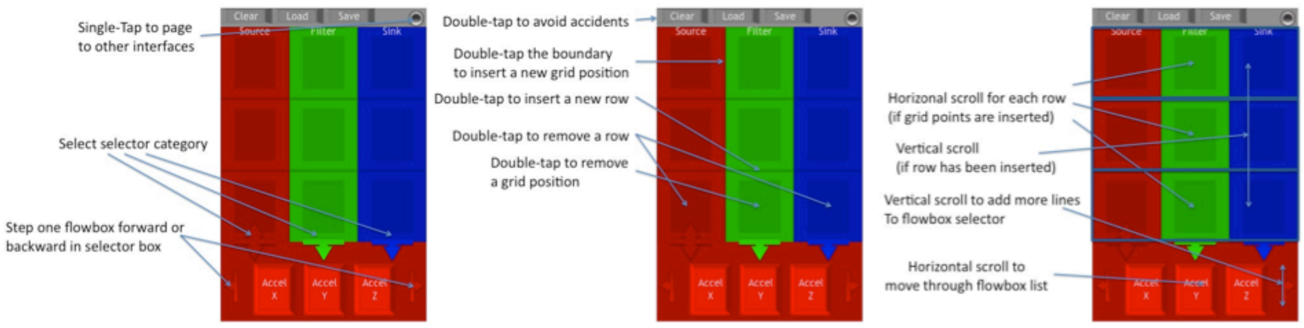

Single-Tap to page to other interfaces
Select selector category
Step one flowbox forward or backward in selector box
Double-tap to avoid accidents
Double-tap the boundary to insert a new grid position
Double-tap to insert a new row
Double-tap to remove a row
Double-tap to remove a grid position
Horizonal scroll for each row (if grid points are inserted)
Vertical scroll (if row has been inserted)
Vertical scroll to add more lines To flowbox selector
Horizontal scroll to move through flowbox list
Clear
Load
Save
Source
Filter
Sink
Accel X
Accel Y
Accel Z

Figure 7 - Interaction functionality of the default urMus interface.

make the relationship between spatial position on the screen and note pitch. The second is visual feedback. We can use these regions for various forms of visual feedback. Animating through various keystroke textures, or simply recoloring the key could achieve this.

The instrumentation of a key looks like this:

```
        whitekey[i]:Handle("OnTouchDown", PlayWhiteKey)
        whitekey[i]:Handle("OnEnter", PlayWhiteKey)
        whitekey[i]:Handle("OnTouchUp", ReleaseWhiteKey)
        whitekey[i]:Handle("OnLeave", ReleaseWhiteKey)
```

This means that the key is sensitive to touch events. It will play a note if a touch presses down or moves into the region (to allow glissando play) and it releases the key if the touch is lifted or moved out of the region. The Handle() method registers functions to respond to these events.

```
function PlayWhiteKey(self)
        local pushflowbox = _G["FBPush"]

        if pushflowbox.instances and
pushflowbox.instances[1]  then

        pushflowbox.instances[1]:Push(whitepitch[self.key
])
            if pushflowbox.instances[2] then

        pushflowbox.instances[2]:Push(daccel*50.0+0.2)
            end
        end
end
```

This is how one could implement the event handler when a note is to be played. It detects if the default urMus engine has instantiated a first and second Push flowbox. If it finds the first, it assumes that it is connected to a control of frequency and hence will push the frequency from a pitch table into the dataflow. If it finds a second, it will also push amplitude data, modified by a force estimated from the accelerometer, into it.

Note that this interface does not specify which algorithm is used beyond the push. Hence if one wants to replace this with another dataflow, that perhaps uses a different synthesis engine or sound, the interface can still be used unchanged. At the same time one could change the look of the interface drastically without modifying the PlayWhiteKey() function, which serves as the point of interface of the instrument. Hence there is clean separation between these two functionalities and in fact both sides can be replaced by alternatives.

4. DESIGNING MOBILE SYNTHESIS AND MEDIA ENVIRONMENTS

The default mapping interface is a successor to SpeedDial, which was a generic musical instrument mapping interface for 12-key touchpad smart phones using the Symbian OS [6]. SpeedDial essentially sought to offer an on-the-fly mapping paradigm, which minimizes interaction steps needed to achieve functional and meaningful mappings while still retaining as much flexibility as possible. The same goals apply to urMus's default interface. However the input capabilities are different. UrMus is designed to work with multi-touch screen smart phones such as the iPhone, which do not come with a hardware keyboard. Hence it is a primary concern to discover how to utilize multi-touch as the primary means for editing.

Practical considerations on the design were:

- Finger size dictates size of interacting object.
- Keep things as large as possible.
- Manage space.
- Be safe if possible (avoid glitches and slips)
- Allow very fast construction of meaningful outcomes

With these considerations in mind we went through a semi-structure design process to arrive at a first interface implementation.

4.1 DESIGN PROCESS

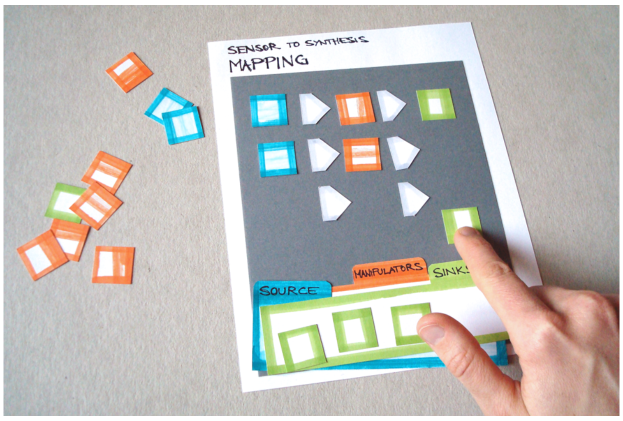


Figure 8: Interactive paper prototype of default synthesis mapping interface of urMus.

In order to prefigure design choices in actual implementation we employed a range of exploratory pre-software design techniques. This has two purposes. First find what the design needs are for the system itself. The second was clarity in design of the early interfaces.

At the beginning we started with quick drawings of the GUI and subsequently they represented a basis for discussion within the team. This method is getting more and more popular in interaction and interface design [1]. Imaginary scenarios were run to select more suitable and intelligible layout concepts for their enhancements.

In the second stage we built *Interactive Paper Interfaces,* which are dynamic lo-fi paper snippets where one sees changes in the sketch depending on the action with movable parts [2]. This allows one to think to temporal interaction scenarios and streamline the steps needed to achieve goals of interactions (Figure 8). By simulating the touch screen a feeling for the flexible set-up was conveyed that furthermore opened up envisioning a range of input principles.

An attempt to anticipate the mobile experience the paper interface is transferred to the iPhone. In addition to manipulating the interface, the physical holding and touching of the device are factors that can be tested in the third stage of the design process [4]. Also lab testing as well as on-the-fly demonstration and observation can be performed. This process already allowed us to narrow down the design decisions substantially, and the paper prototype depicted in Figure 8 already offered many conceptual capabilities that later made the default mapping interface of urMus.

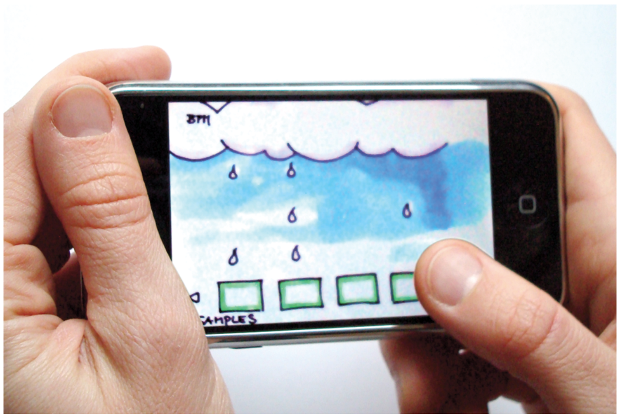


Figure 9: Sketch-in-screen prototype of a raindrop sequencer.

Supported by basic software functionality and through tactile interactivity on the screen we could enhance the discussion on usability providing a realistic performing experience. We also use this process to prototype specific instruments. Figure 9 shows a sketch of sequencing interface based on a raindrop metaphor. By scanning the hand sketch and importing it into urMus one can already start instrumenting interactions by overlaying the image with hidden yet interactive regions and test if the sketch works as intended when interactive.

4.2 Implementation of the default urMus mapping interface

The urMus default interface serves as one of the most developed conceptual prototypes to illustrate the power of the urMus 2D layouting engine. Virtually all layouting capabilities are used here. The interface is inherently fully multi-touch and the number of supported touch points is only limited by the device's capabilities (5 for iPhone/iPod touch, 11 for iPad). Overall this is all implicit and provided by the engine. At the Lua level the programmer can ignore the multi-touch handling. Events that are relevant to regions will automatically be directed to the region independent of the finger used. Hence activating, dragging, and scrolling can all be performed in parallel on multiple regions. At the same time as many interaction elements as possible are kept at a size that can be operated by finger tapping and sliding.

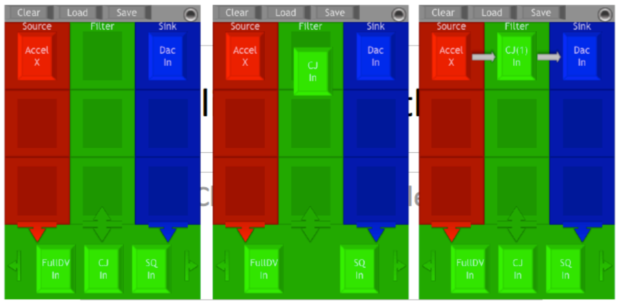


Figure 10 - Creating a flow by grid placements. Connections are established automatically.

Functional flows are established by horizontal arrangement. Flowboxes can be moved from the selector area at the bottom of the screen into grid positions. If they are released in the neighborhood of one of those grid positions they snap into it. If there is a neighboring flowbox left or right of the block it will automatically connect the blocks and establish a flow. If it is connected to a valid sink output will be generated. This process is depicted in Figure 10. This allows for very rapid mapping. In fact one can quickly tap a flowbox to link and unlink an element. We consider this a form of live mapping. By connecting blocks implicitly one removes addtional editing steps such as drawing the connection.

5. FUTURE HARDWARE AND INTERFACE DESIGN

UrMus is very much designed to be platform agnostic. While currently very much inspired by the revolution of the mobile smart phone platform it is intended to be persistent through the rapid changes in hardware that has been characteristic of the mobile hardware development of the last years. This is another reason why urMus does not provide a canonical interface solution. Rather it offers ways to implement many different solutions, which can be tailored towards the strengths of each individual platform. The current architecture should easily support interaction paradigms on emerging table or ePaper hardware and is intended to be extensible to new hardware sensor and actuator capabilities. We see urMus not so much as a specific software solution, but an environment that will support on-going and evolving research in musical instrument and musical environment design on various computing platform.

6. CONCLUSIONS

In this paper we discussed urMus as an environment to design both mobile music instruments and general mobile synthesis environments. UrMus is a meta-environment meant to support UI and interaction design at multiple levels while also supporting key interactive functionality inherently, such as moving, scrolling or resizing regions of a user interface. It allows full support of all sensory capabilities of the device. Currently urMus only runs on iPhone SDK-compatible hardware, but we plan to extend the support to other platforms. Also the number of developed interfaces are currently limited and we hope that with wider use, there will be a larger chest of suggested interfaces to choose from, or to modify.

UrMus can be found at:

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

7. ACKNOWLEDGMENTS

Many thanks to the students of the Fall 2009 course "Building a Mobile Phone Ensemble" at the University of Michigan, who suffered through and added numerous interface and instrument designs to urMus. Also thanks to numerous parties who offer open source solutions that helped speed up the development of some functionality found in urMus. urMus is inspired by ideas found in Blizzard's World of Warcraft addon API and uses the fast embeddable script language Lua.

8. REFERENCES

[1] Bolchini D, Pulido D, Faiola A. “Paper in Screen” Prototyping: An Agile Technique to Anticipate the Mobile Experience. In: Interactions XVI, 2009:29-33.

[2] Buxton, B: *Sketching User Experiences.* Morgan Kaufmann, San Francisco, 2007.

[3] Dannenberg, R. “The Nyquist Composition Environment: Supporting Textual Programming with a Task-Oriented User Interface,” in *Proceedings of the 2008 International Computer Music Conference,* San Francisco, CA: The International Computer Music Association, August 2008.

[4] De Sà, M., Carriço L.: *Low-fi prototyping for mobile devices.* In: CHI '06 extended abstracts on Human factors in computing systems - CHI '06. New York, USA.

[5] Eaglestone, B., Ford, N., Holdridge, P., and Carter, J., “Are Cognitive Styles an Important Factor in the Design of Electroacoustic Music Software?,” Proceedings of the 2007 International Computer Music Conference, International Computer Music Associaation, (2007), pp. 466-473.

[6] Essl, G. “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] Essl, G. “UrMus – an environment for mobile instrument design and performance,” In Proceedings of the International Computer Music Conference, 2010.

[8] Essl, G. “UrSound – live patching of audio and multimedia using a multi-rate normed single-stream data-flow engine,” In Proceedings of the International Computer Music Conference, 2010.

[9] Garton, B. "[chuck~]". Available at http://music.columbia.edu/~brad/chuck~/.

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

[11] Magnusson, T. 2007. "The ixiQuarks: Merging Code and GUI in One Creative Space." In *Proceedings of the International Computer Music Conference*. San Francisco: International Computer Music Association. 2: 332-339.

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

[13] Mrmr, Technical documentation available at http://poly.share.dj/projects/#mrmr, retrieved on January 20, 2010.

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

[15] Puckette, M. “Pure data: Recent progress,” in Proceedings of the Third Intercollege Computer Music Festival, 1997, pp. 1–4.

[16] Puckette, M. “Max at seventeen,” Comput. Music J., vol. 26, no. 4, pp. 31–43, 2002.

[17] RjDj, Techical discussion available at: http://trac.rjdj.me/, retrieved on January 20, 2010.

[18] Shneiderman, B. “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.

[19] Simone, C., Divitini, M. and Schmidt, K. “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.

[20] Villar, N. and Gellersen, H. “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.

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

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

[23] Wang, G. Misra, A. and Cook, P. R. “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.