URSOUND – LIVE PATCHING OF AUDIO AND MULTIMEDIA USING A MULTI-RATE NORMED SINGLE-STREAM DATA-FLOW ENGINE

Georg Essl

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

ABSTRACT

UrSound is a multi-rate normed single-stream data-flow engine for audio and multimedia I/O designed for multi-touch sensor-rich mobile devices. All components connectivity is treated equally. This design choice necessitates that no entity dictates the sample rate and that the semantics of a connection cannot be fixed. Hence we are lead to introduce a multi- and flexible-rate data flow that uses *normed data streams* to allow for seamless and very rapid connectivity changes. We introduce canonical semantic mappings to allow each processing unit to define the meaning of the normed data for their purposes.

1. INTRODUCTION

Mobile phones have become attractive platforms for audio and multimedia processing. There is a need for a audio and multi-media dataflow engine suitable for this platform. This paper presents UrSound, a multi-rate single-stream data-flow engine. It is part of UrMus which additionally offers a scripting API, interface design and other components to help support mobile interactions beyond data flow. Our purpose here is to discuss only the data flow and details of UrMus are described elsewhere [6]. UrMus is a followup to the SpeedDial [5].

Audio processing engines have a long-standing history going back to its origins with Music I by Max Matthews. Ultimately multiple paradigms have emerged addressing how to allow users to generate and process music. The most dominant paradigms are text-based systems, such as CSound [1], Arctic/Nyquist [4, 3], SuperCollider [8] or ChucK [13] on the one hand, and graphical patching systems, such as Max/MSP [10] or pure data (pd) [9] on the other hand. The question of mapping has been widely discussed for these different systems. Mapping of this kind is also of interest for audio analysis, and in this context an approach called implicit patching [11] was proposed by Tzanetakis and coworkers. UrSound allows implicit patching, but does not demand it.

The main difference of UrSound to prior art is two-fold. First UrSound is a data-flow processing pipeline which treats all flow of data equally. Because sensors and actuators run at different natural data rates or at irregular time events this immediate leads to the need for the engine to be multi- and flexible-rate. The second difference is the design goal of *live patching*. This means that elements in the data flow can be repatched interactively. In order to enable this UrSound introduces *normed data*. The outcome of removing semantics from processing allows fast live patching without the performer being required to know which mappings have fitting semantics.

UrSound does not directly assume how it is being used in terms of visual representation. As we will see it can be used on two levels. One is directly in C/C++/Objective-C and embedded in arbitrary projects. In this form it is akin to the core engine of pure data [9]. It can also be used from an abstracted higher level script language Lua as part of UrMus [6]. The choice of Lua is not particularly important for UrSound, except to say that UrSound is not, like for example ChucK, SuperCollider or the Lua-based vessel [12] a script language to write synthesis algorithms. Instead it is purely a script-exposed interface that allows to create C-level connections between processing blocks and hence establish data-flow networks. This is somewhat related to mechanisms also found in Nyquist [3] and Marsyas [11]. The reason for this design choice is purely based on performance. By avoiding any script language processing within the audio pipeline all code ultimately is C, and by avoiding a virtual machine or byte-code processing, one can avoid on-the-fly overhead when reconnecting components.

2. GENERIC INTERCONNECTIVITY AND EQUALITY OF FLOWBOXES

UrSound is a dataflow engine consisting of a network of connected units. When the network connects to an actuator there will be a perceptible outcome. When it connects to a sensor, there will be interactive control. The way UrSound differs from some earlier engines is in the way sensors, actuators and the data flow is treated. Currently UrSound does make one assumption, which is that the data flow is just one normed data point in the range of $[-1, 1]$. Hence currently the engine does not allow higher dimensional information to be transported as a single object.

In UrSound we do not distinguish between types of data streams. This is different to many other engines which distinguish certain data flows by function. In those cases unit generators are often seen as having two types of data flowing in and out of them. In pure data for example these two types are described as message and signal, other place they are described as control and signal.

There are numerous advantages to removing this distinction. One is that it leads to equal and consistent treatment of all parts of the network. For example should a signal coming out of a microphone be considered a control/message stream, or should it be considered a signal? Should a new touch event on a multi-touch screen be considered a control or a signal? By removing the distinction the question does not pose itself. Rather the network itself is allowed to define the semantics of a data flow. An event is simply a data stream that is currently sending data. A sequence of events is a data stream which sends data at irregular event times.

Looking at unit generators we can make this more explicit. Let us take a unit generator that is a sine oscillators:

$$y = A \cdot \sin(2\pi f \cdot t) \tag{1}$$

with the following inputs: amplitude A, frequency f, and t is some notion of time. We have one output which is the result of the oscillator given its parameters. Traditionally we consider A, f to be controls that generate a signal stream y.

But what is the correct way to treat the following goal in this context: Have a microphone signal (which traditionally is an audio signal stream) change the amplitude and have accelerometer data change the frequency. If the microphone signal is not a control, how do I convert it to serve as control to the unit generator? In UrSound we remove this question and design the necessary aspects that are needed to make this work.

3. NORMED DATAFLOW

If one removes the difference between control and signal one immediately can consider connecting every data flow with another. But by wanting to do so there are two problems that need to be addressed. These problems relate to the question of data semantics and timing.

To understand what we mean by data semantics let us return to the example of the sine oscillator. As written in equation (1) has inputs A, and f. These however require different actual number ranges. To have a sine oscillator that does not clip or overflow, the absolute value of A should be no larger than 1. On the other hand f in order to be in the audible range should be somewhere between 20 and 20000. Another way to think about this is via types or units. The unit of frequency is Hertz. We do not typically give amplitude of audio samples a type though we could. Secondly the data itself has a certain semantics. It is actually not enough to know that the range of frequencies is 20 to 20000 but in addition the perception of frequencies mean that our perception of pitches relates to frequency doubling, hence having an exponential relationship. On the other hand the amplitudes A are generally treated linearly.

The goal is now to make it easy for an arbitrary data stream to be both connectable to A and f and make that give sensible results. The solution used in UrSound is twofold. First all data streams are *normed*. By this we mean that their value range is generically assumed to be within $[-1,1]$. Unit Generators which observe this condition are called *flowboxes*. Any flowbox in UrSound is required to accept this range for all its inputs, and produce outputs that satisfy this requirement. This alone already allows for generic interconnectivity. Now any flowbox output can be connected with any flowbox inputs without consideration of the type or semantics of the input. The second part of the solution has to do with semantics and type. UrSound flowboxes are responsible for knowing their own semantics and type, and be capable of converting a normed data stream into semantic data that is meaningful for its own purpose.

In the case of the sine oscillator flowbox, the generation of the semantics of frequency requires a mapping of the interval $[-1,1]$ to some frequency range. This mapping is somewhat arbitrary and up for design of a flowbox. In order to ease the design, we use a set of canonical mappings in UrMus. If a flowbox has an input of a certain frequently used semantics there is one defined mapping from normed data to this input. Frequency is a good example of an input that appears in a wide range of contexts. The currently used canonical mapping for frequency is $out = 55 \cdot 2^{96*in/12}$. This allows for sub-audible frequencies to be represented in one mapping hence allows for effects that happen in the visual range as well as time-domain audio oscillations (such as vibrato) to be rendered with the same mapping.

The choice of the interval $[-1,1]$ is somewhat arbitrary and in fact when a data network is connected it is mostly invisible what the data is that flows through the network. There are however certain reasons that suggests this choice. For one it is a good prototype for a symmetric bound interval. Many types of data that we encounter in multi-media processing are either symmetric bound, or asymmetric bound. Examples for symmetric bound data are audio samples, or tilt angle from accelerometers. Second, the interval is general in the sense that any function, even infinitely continuous ones can be arranged with a suitable mapping. In the case of an infinite line, it can be defined through a stereographic projection. Also the interval is the natural parallel projection of the circle hence rotations on the circle can be naturally done using this interval. This has direct practical applications in that for example the zero-point of data from accelerometer tilt can be directly manipulated. Finally within the interval $[-1,1]$ monomials such as x^n all have the property that they all share intersections with the 0 and ± 1 points hence increasing the order serves to simply create increasingly steep even and odd-symmetric function that for very high order start to approximate steps with increasing flattening around 0.

4. MULTI-RATE DATAFLOW

UrSound is a multi-rate as well as flexible-rate single-dimensional dataflow pipeline. By multi-rate we mean that different parts of the dataflow network may be operating at different data rates. Some parts of the dataflow network may also operate at irregular rates. The necessity for this again follows from removing the distinction between control and signal. Of course one could try to design a system where one data rate plays a global role. In fact many audio synthesis engines take this position indirectly, because they do not really consider other output media, hence it is fine to consider a global sample rate. In UrSound inherently we wanted to design for a number of existing data sources and data sinks.

The idea is that sub-graphs of the dataflow network take on the data rate that is natural to its connected components, insofar as this is well-defined. Take again example of the sine oscillator (1). If we connect an input to the accelerometer, the data rate of the network consisting of the accelerometer leading into the input operates at the natural rate of the accelerometer. If reconnected to a microphone, then the natural rate should be that of the microphone.

Furthermore the sine oscillator also serves to illustrate another important question. How do we identify which parts of the data-flow network have what sample rate? Clearly the rate of data at the output of the sine oscillator is not dependent of the rate that changes any of the inputs. Hence the sine oscillator is an example of a rate-decoupling flowbox. Many traditional unit generators have this property. However there is a second type of flowbox that does not de-couple rate. All filters fall into this class. The rate at the input of the filter generically is related to the rate at the output, at least form some input/output pair. So in general a flowbox may have input/output pairs that are decoupled as well as those that are coupled. If a dataflow network is connected to a coupling pair, then the rate of the network will propagate through the flowbox. If it is decoupled it will terminate at the flowbox.

With these rules we can already describe the data rates of subgraphs of many data flow networks, and in fact most networks that one encounters in typical synthesis settings. There is however an additional case to consider. If a subnetwork connects two decoupled flowboxes the network inbetween may not have any inherent sample rate. Currently UrSound does not offer a canonical resolution for this situation. There are three options that can be implemented: (1) take the rate of the incoming decoupled flowbox, (2) take the rate of the outgoing decoupled flowbox, (3) define an independent sample rate for the subnet.

The data rate can propagate forward, that is from an input to an output, or it can propagate backward, from an output to an input. In UrSound this propagating relation is explicitly chosen because this allows for user-side resolution of conflicting possibilities. For example assume that Accelerometer data is directly connected to the audio output. Each has its natural rate. How can one define which rate will be used in data flow? The answer is by choosing the direction of the flow. In UrSound parlance we call a situation a *push* if the data rate is defined by a flow from an output (say the accelerometer) to an input (say the dac). The opposite case is called a *pull*. Hence either the dac can pull data from the accelerometer at its rate, or the accelerometer can push data to the dac at its rate. All sources and sinks inherently know rate conversion hence it is always possible to push or pull them.

5. FLOWBOXES

The simplest possible flowboxes only have one input, one output and do not de-couple data rates and have no state. Within UrSound we call these flowboxes *atomic*. Within audio processing we know flowboxes of this type as waveshaping [7]. However within UrSound they immediately have a multitude of meanings depending on where they are used in the network. They only become waveshapers if their input is an audio signal. Otherwise they have more the nature of a signal manipulator.

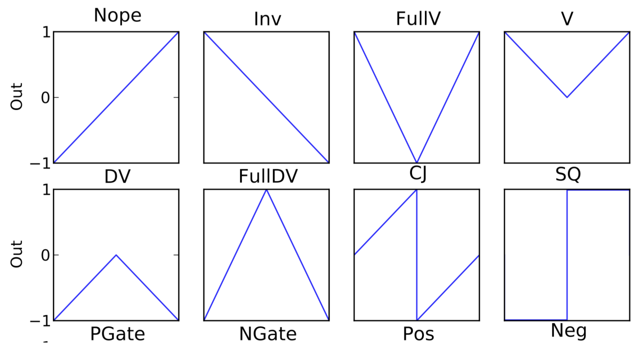

Figure 1. 8 of the most elementary (piecewise linear) atoms. UrMus also uses polynomial, exponential, logarithmic and trigonometric atoms.

UrSound comes with a fairly large set of atomic flowboxes to ease data manipulation for many likely cases. The most simple examples are depicted in Figure 1. The trivial atom is `Nope` which does nothing to the signal. Many of these are motivated by certain need to change the reference or semantics of sensor signals. An important semantics is orientation and the position of the zero. For example `Inv` inverts the signal, hence an up-down tilt becomes a down-up tilt. `V` defines that the minimum should be at 0 of the mapping rather than at -1. This has the effect of pulling the rest position to the center of the mapping. `DV` does the same but for the maximum. The line $[-1, 1]$ has a jump condition if one periodically continues the interval. The `CJ` atom moves that jump condition to 0 and hence acts as a 90-degree rotation. It can also do signal manipulations. For example `SQ` forces all positive values to 1 and all negative values of -1 hence yielding a binary oriented signal from any continuous one. Another specialized flowbox is called *ZPuls*. Its purpose is to convert special cases into a discrete event. In this case it returns 1 if it sees a 0 in the incoming stream, hence pulsing for this special case. One can envision a wide range of pulse generating methods that all generate pulses whenever conditions on the input signals are met. This is the way conditionals can be implemented within the data network.

STK [2] is used to provide a range of more complex flowboxes. The paradigm of STK is not fully compatible with a multi-rate system and often inputs to STK algorithms are not all of a type that seamlessly would operate with within the type of data flow mapping the UrSound uses. This specifically goes for consistent behavior of inputs with same or similar semantic meaning. However, the current implementation is workable and shows how complex algorithms can be incorporated into the system.

6. CONCLUSIONS

UrSound is a dataflow engine that works with multiple frame rates and operates on normed data on the range of $[-1, 1]$. FlowBoxes, a generalized form of unit generators provide conversion of the normed data to semantically meaningful information. Hence control and signal become fully interchangeable. UrSound serves as an on-the-fly patching multimedia dataflow backbone to the UrMus environment. However, in principle it can be replaced by other environments that offer sound and multimedia processing. We can see that currently developed alternative solutions such as pd, ChucK can be integrated to either be concurrent or replace UrSound for certain purposes. Regardless we believe that the the multi-rate and normed data flow design of UrSound is attractive and live-patching is important for interactive music generation. Even though UrSound was designed on mobile smart-phones such as the iPhone, it can in principle be used on any platform that requires a dataflow mapping solution.

7. ACKNOWLEDGEMENTS

I appreciate stimulating discussion with George Tzanetakis on implicit patching.

8. REFERENCES

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

[2] P. R. Cook and G. P. Scavone, "The synthesis toolkit (stk)," in *Proceedings of the International Computer Music Conference (ICMC)*, 1999.

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

[4] 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.

[5] 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.

[6] ——, "UrMus – an environment for mobile instrument design and performance," in *Proceedings of the International Computer Music Conference (ICMC)*, Stony Brooks/New York, June 1-5 2010.

[7] M. Le Brun, "Digital Waveshaping Synthesis," *Journal of the Audio Engineering Society*, vol. 27, no. 4, pp. 250–266, 1979.

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

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

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

[11] 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.

[12] 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.

[13] 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.