Yeah, ChucK It! => Dynamic, Controllable Interface Mapping

Ge Wang* gewang@cs.princeton.edu
Ananya Misra* amisra@cs.princeton.edu
Ajay Kapur▼ ajay@ece.uvic.ca
Perry R. Cook*† prc@cs.princeton.edu

*Department of Computer Science †(also Music)
Princeton University, Princeton, NJ, U.S.A.

▼Music Intelligence and Sound Technology Interdisciplinary Center (MISTIC)
University of Victoria, Victoria, BC, Canada

ABSTRACT
ChucK is a programming language for real-time sound synthesis. It provides generalized audio abstractions and precise control over timing and concurrency - combining the rapid-prototyping advantages of high-level programming tools, such as Pure Data, with the flexibility and controllability of lower-level, text-based languages like C/C++. In this paper, we present a new time-based paradigm for programming controllers with ChucK. In addition to real-time control over sound synthesis, we show how features such as dynamic patching, on-the-fly controller mapping, multiple control rates, and precisely-timed recording and playback of sensors can be employed under the ChucK programming model. Using this framework, composers, programmers, and performers can quickly write (and read/debug) complex controller/synthesis programs, and experiment with controller mapping on-the-fly.

Keywords
Controller mapping, programming language, on-the-fly programming, real-time interaction, concurrency.

1. INTRODUCTION & MOTIVATION
Mapping of real-time sensor input to control audio synthesis is an important part of computer music performance and research. An expressive programming system for controller mapping must accommodate a wide variety of input/output modalities that can be connected to control audio synthesis and other processes. Yet, the underlying programming paradigm should be flexible enough to specify any (Turing) programmable task. Furthermore, the specification (code, flow graph, etc.) should be easy to write and read/debug, even when the program has gained considerable complexity. Finally, the system implementation should be optimized to achieve robust real-time performance.

Low-level languages like C/C++ are powerful and expressive in that the programmer can specify all the details of a system. But the drawback is that even simple tasks can be cumbersome to program, even with careful abstraction, such as in the Synthesis ToolKit (STK) [2]. High-level computer music languages like Max/MSP [11], Pure Data [12], SuperCollider [10], and others [1,3,4,13] provide the ability to quickly prototype a system, but rely heavily on existing modules (and often less on the language itself). Furthermore, time and timing in many of these systems are not apparent when writing, viewing, or running a program.

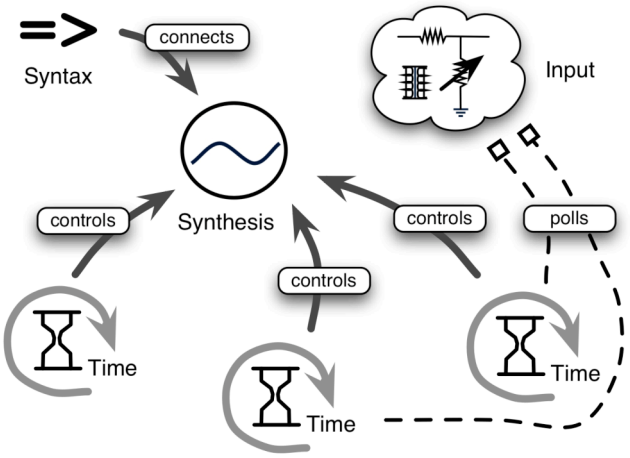

Figure 1. The ChucK Programming Model. Data flow (synthesis) and timing flow (control) is fundamentally separated. ChucK operator (=>) connects unit generators, while precise timing is specified in one or more shreds (represented here by hourglass loops) to collect input data and to control sound synthesis.

ChucK [14] attempts to combine the rapid-prototyping advantages of high-level languages such as Pd with the expressiveness of lower-level languages, such as C/C++. To achieve such a balance, ChucK abstracts only data (i.e. in unit generators) and exposes full control over time and concurrency in the syntax and semantics of the language (Figure 1). Programmers can quickly specify a patch (network of unit generators), as in Pd - but without any implicit control rate timing. The ChucK programmer can then specify the timing in an imperative manner, by combining control structures (e.g. if, else, for, while, etc.) with precise timing directives (e.g. 80::samp => now, or "advance time by 80 samples"). Furthermore, it is possible to have many parallel timing specifications precisely synchronized to each other. The resulting framework makes timing both easy to specify and to read/debug, and forms the foundation for our time-based controller-mapping paradigm.

Based on the timing and concurrency mechanism, it is straightforward to write programs that read sensor data and map

them to sound synthesis parameters. Input-polling and event synchronization can be placed at any point in time. Control rate can be dynamically specified (or computed) for each sensor (e.g. FSR's every 5 milliseconds, accelerometers every 50 milliseconds, temperature sensors every 30 seconds). Concurrency allows for any number of sensors to be handled separately in an organized and scalable manner.

The remainder of this paper is organized as follows. *Section 2* describes the basics of using ChucK to map sensors to control sound synthesis. *Section 3* covers some advanced features, such as dynamic/on-the-fly patching, dynamic voice allocation, concurrent sensor mapping, and recording + playback of sensor data. *Section 4* describes some case studies for controller mapping with ChucK. Finally, we discuss future work and conclude in *Section 5*.

2. BASIC MAPPING IN CHUCK

The ChucK framework makes it easy to write a program that maps controller input to sound synthesis. The syntax has been described extensively in previous works [14-16]. In this section, we present simple examples that illustrate basic tasks such as mapping MIDI input to synthesis parameters, using both polling as well as non-polling, event-based techniques. Although the examples focus on MIDI input, the framework can be easily modified for other standards (e.g. network I/O, serial I/O, and Open Sound Control [17]), sensors, as well as new algorithms for sound synthesis.

2.1 A Simple Polling Example

The mapping process can be divided into two components: setting up a synthesis patch, and specifying the timing information for controlling it.

```
01# // the patch: sine wave to reverberator to dac
02# sinosc s => JCRev r => dac;
03# // initial settings
04# .2 => r.mix;
05#
06#  // declare a MIDI message to be filled in
07#  MidiMsg msg;
08#
09#  // infinite time-loop
10#  while( true )
11#  {
12#     // advance time (by 5 milliseconds)
13#     5::ms => now;
14#
15#     // poll MIDI - (could be any event)
16#     while( midi.in( msg ) )
17#     {
18#        // if the message was a note on
19#        if( msg.type == midi.NOTE_ON )
20#        {
21#           // convert to frequency, and set
22#           std.mtof( msg.data1 ) => s.sfreq;
23#           // set gain of sinosc
24#           msg.data2 / 128.0 => s.gain;
25#        }
26#        else if( type == midi.KNOB )
27#        {
28#           // set dry/wet mix and gain of reverb
29#           midi.data1 / 128.0 => float mix => r.mix;
30#           .25 + .25 * (1.0-mix) => r.gain;
31#        }
32#     }
33#  }
```

Figure 2. A simple skeleton framework for polling input to control sound synthesis.

A sound synthesis patch can be quickly constructed by connecting unit generators, using the ChucK operator (=>). Line 1 (Figure 2) sets up such a patch by connecting a sine oscillator to a reverberator, which is connected to the digital/analog converter. All patches deal only with audio data and have no notion of time or control. This is advantageous because it reserves the ability to specify timing to the programmer.

Timing information is specified in time loops, such as the one on lines 10-33. We can specify any arbitrary timing pattern by advancing time appropriately. Line 13 in the example shows time being advanced by 5 milliseconds. Control over the unit generators is interlaced with timing, allowing the programmer to manipulate them at any time with sample-level precision.

Next, we can collect controller information – in this case, by polling. In our example, MIDI is polled (line 16) for input whenever time is advanced. Because we can specify arbitrary timing patterns and poll for input at any point, the timing pattern implicitly becomes the control rate for the sensor. This gives the programmer the ability to adapt code to poll different sensors at different control rates as needed. For instance, (as mentioned earlier) the ideal control rate for a force-sensing resistor is on the order of milliseconds, whereas a temperature-based sensor may need to be polled every few seconds or minutes.

Once we have obtained control data from polling, we can apply the information immediately or store it for use at some later time. In the example, the frequency, gain and mix parameters of the `sinosc` and `JCRev` unit generators are changed at once based on the specific input received (lines 19-31). However, it is possible to perform a variety of other actions including:

- Dynamically modify the patch by adding or removing unit generators (discussed in *Section 3.1*)
- Spork a new process/shred (using the `spork ~` operator)
- Spork a shred from file (using `machine.add()`)
- Clone a shred (with a `shred.clone()` function)

Furthermore, all the above steps can be carried out concurrently with multiple processes. In this way, many sensors can be read at the same time or information from a single sensor can be used by several processes, through user-defined broadcasts and events (not shown).

2.2 Alternative to Polling: Events

An alternative to polling is to obtain input data using events. Events are system or user-defined objects that a ChucK process can wait on. In this case, time is advanced only when a MIDI event occurs (line 5) – the event is triggered internally by the I/O subsystem. The relevant sensor information associated with that event is passed back via the msg variable, which the process handles as it deems fit. The final translation of sensor data to sound is carried out in the same way as before.

```
01# // infinite time-loop
02# while( true )
03# {
04#   // event (instead of explicitly advancing time)
05#   midi.event( msg ) => now;
06#
07#   // at this point, assume event occurred
08#   // handle msg as desired…
09# }
```

Figure 3. The previous example modified to use events instead of polling.

Events and polling are two methods of achieving the same task. A programmer would use polling when they want precise control over input timing. On the other hand, a programmer might use events in situations where it is appropriate to relinquish some timing control to the system.

3. ADVANCED FEATURES

In addition to the basic methods to map input to synthesis parameters, it is also possible to take advantage of some advanced features as part of the ChucK language. These include dynamic patching, dynamic voice allocation, concurrent sensor mapping, and precisely-timed recording + playback of sensor data. Additionally, it is possible to experiment with these features 'on-the-fly' - without restarting or stopping the virtual machine, leveraging the on-the-fly programming features of ChucK.

3.1 Dynamic Patching

Dynamic patching refers to constructing and modifying synthesis patches at runtime [8]. Because the ChucK patching semantics are inherently dynamic (unit generators can be created, connected, or disconnected at any point in time), this is simply a consequence of the language semantics. For example, it is straightforward to write a program that changes the unit generator graph topology using input from sensors. It would use a similar framework as the example in *Section 4*, but instead of changing unit generator parameters, operations on the unit generator graph itself can be performed (Figure 4).

```
01# // inserting a new unit generator
02# if( switch_hit )
03# {
04#    // disconnect foo from bar w/ 'unchuck' operator
05#    foo =< bar;
06#    // reconnect foo to new Filter to bar
07#    foo => Filter f => bar;
08# }
                          (a)

01# // connect/disconnect two unit generator graphs
02# if( toggle ) pa => pb; // connect
03# else         pa =< pb; // disconnect
                          (b)
```

Figure 4. Two dynamic patching examples. (a) Inserting a new unit generator on demand. (b) Connecting and disconnecting two existing unit generator sub-patches.

For example, Formant Wave Functions (FOFs) can be flexibly controlled in this manner. The number of FOFs needed at a given time depends on the length of the FOF and the current pitch. In ChucK, the programmer can dynamically allocate and delete FOFs by adding or removing unit generators.

3.2 Dynamic Voice Allocation

Dynamic voice allocation allows for entire synthesis patches to be incorporated into the global unit generator graph, sample-synchronously, and without pre-allocation. This is useful for synthesizing polyphony with an arbitrary number of voices. In order to synthesize polyphony, a separate instance of a synthesis patch must be allocated for each voice. Some systems have a fixed maximum number of voices allocated beforehand, which limits the program during runtime, and potentially incurs unnecessary memory and CPU overhead.

In ChucK, a new voice is simply given its own process, also called a shred. A shred can be instantiated on demand and can, in turn, carry out dynamic patching as described in *Section 5.1*. New shreds can be created and assimilated into the virtual machine in the following ways:

```
machine.add( "voice.ck" ) => shred id;
// or...  me refers to this process
machine.add( me.clone() ) => shred id;
```

Figure 5. Dynamic voice allocation can be implemented by inserting a new shred into the VM, either from file or by cloning an existing shred.

There is no preset limit to the number of shreds that can be dynamically instantiated. Shreds/voices can be created and deleted precisely as needed, without pre-allocation.

3.3 On-the-fly Controller Mapping

While it is possible to write programs before execution, the features of on-the-fly programming [15] can be employed to modify and augment the controller mapping logic and parameters during runtime. For example, new program modules can be integrated into the virtual machine, introducing new mappings dynamically without having to restart the system. It is also possible to replace an existing controller mapping or synthesis algorithm by swapping existing shred(s) with updated ones in a sample-synchronous manner. This allows programmers to rapidly experiment with their controller mapping.

3.4 Precise Recording/Playback

A key ChucK tool for successful mapping is the ability to record sample-accurate, synchronized audio and gestural data, enabling a user to record a performance and then playback all the data, tweaking parameters to obtain desired effects and sound synthesis. This is made possible by ChucK's precise access to "now", along with easy abstraction of control and audio data as arrays of time-stamped events. Playback is made easy by iterating through arrays of recorded data and asserting control at each element, and advancing time to the next point using the recorded time-stamps.

4. Case Studies

ChucK proves to be a useful tool with off-the-shelf controllers as well as custom-built instruments. It is easy to have an armada of synthesis techniques at the touch of a command line, from FM synthesis, wavetable synthesis, to control of physical models. ChucK makes it easy to switch between algorithms, manually and programmatically *sporking* new processes and removing others. One fun application is to convert arrays of sliders and knobs (like those seen on the Edirol PCR series) into a DJ beat station using ChucK's inherent timing capabilities.

We also explored with mapping one of the earlier controllers, the Radio Drum [9]. We were easily able to convert all patches written for other case studies, having the drum controller sonifying physical models to FM synthesizers. A successful mapping with the Radio Drum controlled digital audio effect parameters in 3D space, such as reverb and pitch shifters.

Our next experiments used ChucK with Indian drum controllers, the ETabla [5] and Edholak [7]. The ETabla mapped physical models of the tabla [5], which are now ported to ChucK and work well in real time. We also developed a system for high-level feature extraction of input data. This allowed the performer to tap a beat, and ChucK would record the incited rhythm with a chosen

sample or sequence, loop it, transform it, and spork a new process when another rhythm was performed.

Experiments with the ESitar controller [6] using ChucK opened many new avenues for experimentation. As described in [6], the controller takes both the natural audio signal from the traditional instrument as well as control data from a variety of sensors and uses both streams to process and synthesize sound. Using ChucK program modules, we recorded synchronized audio and controller input and then played back the streams in ChucK to tweak parameters on-the-fly in the mapping algorithms until the desired effect was obtained. Experiments with comb-filters, delay lines, ring modulation, and controlling moog synthesizers all proved to be successful and straightforward to implement and control.

Figure 6. ESitar controller using ChucK to communicate with Trimpin's eight robotic turntables.

An exciting experiment was to use ChucK to communicate with Trimpin's eight robotic turntables using the ESitar controller (Figure 6). Trimpin's turntables accept MIDI input, and control motors to turn on, turn off, reverse, forward, speed up, or slow down the record. ChucK made it simple to map gesture/sensor input from the ESitar to the turntables, and to do the robotics programming on-the-fly.

5. CONCLUSION AND FUTURE WORK

We have presented a time-based programming paradigm for mapping interfaces. The primary advantages of this approach are that timing is easy to specify and that the mapping *itself* is highly controllable. Under this framework, it is straightforward to perform tasks such as dynamic patching, dynamic voice allocation and recording/playback of synchronized audio + control data. We have also described both a polling and an event-based method for gathering input data in real-time. We performed experiments on a variety of musical interfaces (which have all survived).

There are many areas for improvement. For example, due to the sample-synchronous architecture of ChucK, it inherently incurs more overhead than frame-based systems, such as Pd, Max/MSP. This sacrifice has enabled ChucK to achieve a different manner of flexibility. However, optimizations to improve system performance are desired. Other future work includes allowing programmers to rapidly build custom graphical user interfaces.

With ChucK in conjunction with environments like the Audicle [16], we hope to provide a full platform for performance and experimentation with controllers, synthesis, and live code.

6. ACKNOWLEDGMENTS

We would like to thank the growing ChucK community for their ideas, discussions, sample programs, and overall support. Special thanks to Adam R. Tindale for his support, ideas, and implementation for this paper. Also thanks to Trimpin for the use of his robotic turntable for experimentation with ChucK.

`http://chuck.cs.princeton.edu/`

REFERENCES

[1] Boulanger, R. *The C Sound Book*. The MIT Press, 2000.

[2] Cook, P.R. and G. Scavone. "The Synthesis ToolKit (STK)," *In Proceedings of the International Computer Music Conference*. Beijing, China, 1999.

[3] Dannenberg, R. B. "Aura II: Making Real-time Systems Safe for Music." *In Proceedings of the International Conference on New Interfaces for Musical Expression.* Hammamatsu, Japan, 2004.

[4] Dechelle, F., Borghesi, R., Ceccco, M.D., Maggi, E., Rovan, B. and Schnell, N. "jMax: a new JAVA-based editing and control system for real-time musical applications." *Computer Music Journal,* 23(3) pp. 50-58. 1998.

[5] Kapur, A., Essl, G., Davidson, P. and P.R. Cook. "The Electronic Tabla Controller," *Journal of New Music Research,* 32(4), pp 351-360. 2003.

[6] Kapur, A. Lazier, A., Davidson, P., Wilson, R.S., and P.R. Cook. "The Electronic Sitar Controller," *Proceedings of the International Conference on New Interfaces for Musical Expression,* Hamamatsu, Japan, pp. 7-12. 2004.

[7] Kapur, A., Davidson, P., Cook, P.R., Driessen, P., and W. A. Schloss. "Digitizing North Indian Performance," *In Proceedings of the International Computer Music Conference.* Miami, Florida, 556-563. 2004.

[8] Kaltenbrunner, M., G. Geiger, S. Jorda. "Dynamic Patches for Live Musical Performance." *In Proceedings of New Interfaces for Musical Expression.* 2004.

[9] Mathews, M. and W. A. Schloss. "The Radio Drum as a Synthesizer Controller," *Proceedings of the International Computer Music Conference*, Columbus, Ohio, 1989.

[10] McCartney, J. "A New, Flexible Framework for Audio and Image Synthesis." *In Proceedings of the International Computer Music Conference*. pp. 258-261. 2000.

[11] Puckette, M. "Combining Event and Signal Processing in the MAX Graphical Programming Environment." *Computer Music Journal.* 15(3), pp 68-77. 1991.

[12] Puckette, M. "Pure Data." *In Proceedings of the International Computer Music Conference.* 269-272. 1997.

[13] Topper, D. "GAIA: Graphical Audio Interface Application." *In Proceedings of the 2004 International Computer Music Conference.* Miami, Florida, 2004.

[14] Wang G. and P. R. Cook. "ChucK: A Concurrent and On-the-fly Audio Programming Language." *In Proceedings of International Computer Music Conference.* pp. 219-226. 2003.

[15] Wang G. and P. R. Cook. "On-the-fly Programming: Using Code as an Expressive Musical Instrument." *In Proceedings of the International Conference on New Interfaces for Musical Expression,* Hamamatsu, Japan, pp. 138-143. 2004.

[16] Wang G. and P. R. Cook. "The Audicle: A Context-sensitive, On-the-fly Audio Programming Environ/mentality." *In Proceedings of the International Computer Music Conference.* 2004.

[17] Wright, M., Freed, A., and A. Momeni. "Open Sound Control: State of the Art 2003" In Proceedings of Conference on New Interfaces for Musical Expression. Montreal, Canada, pp. 153-159. 2003.