

The Audicle: A Context-Sensitive, On-the-fly Audio Programming Environ/mentality

Ge Wang and [†]Perry R. Cook

Computer Science Department, [†](Also Music)
Princeton University

{gewang, prc}@cs.princeton.edu

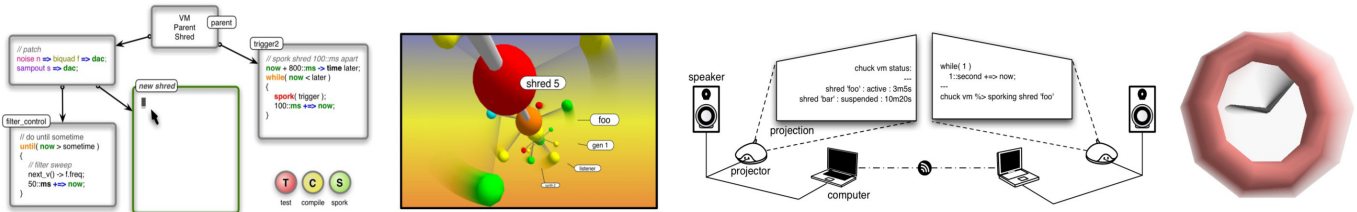


Figure 0. Some things associated with an Audicle.

Abstract

Many software environments have been developed for computer music. Programming environments typically provide constructs to implement synthesis or musical algorithms, whereas runtime environments allow performers to exert parametric control over their programs onstage, in real-time. We present a new class of audio programming environment that integrates the programmability of the development environment with elements of the runtime environment. The result, called the Audicle, is a novel integration of a concurrent “smart” editor, compiler, virtual machine, and debugger, all running in the same address space, sharing data, and working together at runtime. We believe this augmentation has the potential to fundamentally enhance the way we write and visualize audio programs both offline and on-the-fly.

In this paper, we examine existing programming and runtime environments, present the ideas behind the Audicle, and demonstrate its features and properties. Our model of the Audicle is tightly integrated with the Chuck programming language and inherits many of its fundamental properties, including: decoupling of data-flow and time, concurrency, and modularity for on-the-fly programming. We discuss the main components of the Audicle, and show that it not only provides a useful class of programming tools for real-time composition and performances, but also motivates a new type of on-the-fly programming aesthetic – one of visualizing the audio programming process.

1. Introduction

Software environments play a pivotal role in the creation and performance of computer music. Programming environments provide the setting to design/implement synthesis and

compositional algorithms. Runtime environments realize and render these algorithms into sound (and images), and allow performers to interact with the system, often in real-time. In this work, we present a new class of audio programming environment, called the Audicle, which combines the programmability of programming environments and the immediate feedback of runtime environments. The Audicle is an integration of “smart” concurrent editor, compiler, virtual machine, and visualizations, all operating in a single on-the-fly environment. This has the potential to fundamentally enhance the way we write, visualize, and interact with audio programs.

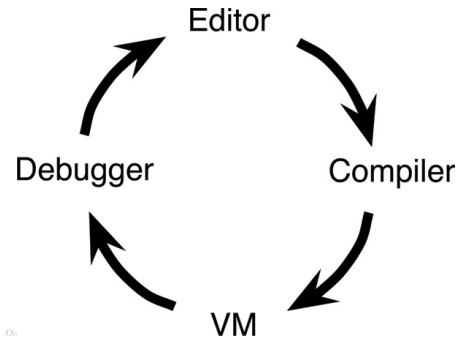


Figure 1. Completing the loop. The Audicle bridges runtime and development by integrating elements of both.

The Audicle differs from traditional environments in the following ways. First, it brings the editor and compiler into the runtime environment (Figure 1), which allows a fundamentally greater level of interactivity in the programming process. Secondly, it is tightly coupled with a programming language – in this case, Chuck, a concurrent and on-the-fly synthesis language (Wang and Cook, 2003). This coupling is advantageous because it leverages and enhances the desirable Chuck properties of precise timing and concurrency. This is

different from systems like Max and Pure Data, where the environment essentially *is* the language. In the Audicle, the environment *complements* the underlying language and enhances the ability to develop and visualize programs both offline and on-the-fly. Thirdly, the Audicle is an aesthetic and mentality of *visualizing the process* of on-the-fly programming and the state of the runtime system.

This paper is organized as follows. In *Section 2*, we present the goals of the Audicle and also examine the background of existing programming and runtime environments, both audio and non-audio. The Audicle’s runtime compiler, virtual machine, and much of the concurrent editor semantics are based on our earlier work with ChuckK and on-the-fly programming. In *Section 3*, We overview their key features and properties, as well as the challenges of on-the-fly programming as they pertain to the Audicle. In *Section 4*, we present the design of the Audicle, introduce the *Audicube*, and discuss its properties and implementation. In *Section 5*, we address how these ideas motivate a new kind of *on-the-fly programming aesthetic*, and discuss future work.

2. Background

2.1 Goals of the Audicle

A simple but important question to answer is: *why investigate the programming environment?* We believe that the programming language and the environment it is used with fundamentally influence how we think about and write programs. ChuckK provided a new way of reasoning about time, data-flow, and concurrency in a programming language. The Audicle is designed to enhance and complement these features, and make them more accessible, faster, and perhaps more enjoyable to use.

Context-sensitive. The editor is to allow concurrent code to be clearly entered and represented. Also, it is to have knowledge of the deep structure of the program and runtime information (such as program state and profiling hints) – and can use this information to aid the programmer to more easily write code. We call this a *concurrent, “smart” editor*.

On-the-fly. On-the-fly programming is the practice of coding at runtime – while the program is running. The Audicle aims to complete the development/runtime loop by bringing the editor and compiler to the virtual machine, and vice versa. By making them accessible to each other, new and faster interfaces and paradigms for runtime audio programming may emerge.

Different views. Having different views of the same program can be very useful to writing and fine-tuning code. The Audicle should allow a program to be viewed and manipulated in many ways: as concurrent code, syntactic/semantic representations, or timing and synchronizations. Additionally, the Audicle is be an observation or visualization of the *process* of on-the-fly programming, giving it the potential to be a useful, general-purpose performance or educational tool.

Minimal. The Audicle is to provide a minimal interface, and rely on the underlying interactions of the language and the multiple viewing models to achieve a great deal of expressiveness and power, without imposing a particular programming style.

2.2 Existing Environments

An environment, in the context of this investigation, is defined as a comprehensive software setting in which programming and/or runtime control is carried out and/or facilitated. There have been many environments developed for programming, performance, and composition, as well as several environments not specifically intended for these purposes that are also useful to examine.

As mentioned, *programming environments* provide a setting to write and edit programs at development time, and often include a compiler and debugger. Examples include graphical environments such as Max (Puckette, 1991) and Pure Data (Puckette, 1996), integrated development environments (IDE’s) for text-based languages such as Java, C/C++, Nyquist (Dannenberg, 1997), and SuperCollider (McCartney, 1996), and software frameworks such as Ptolemy (Lee et al. 2003). These environments allow code, flow graphs, and other programming constructs to be entered, compiled, and run (in separate phases).

Runtime environments, on the other hand, provide an engine and a related set of interface elements for manipulating parameters at runtime (and often in real-time). The graphical front-ends of Reaktor and Max, as well as Real-Time CSOUND (Vercoe, 1990), Aura (Dannenberg and Brandt, 1996), and more recently, Soundium 2 (Schubiger and Muller, 2003) are good examples of runtime environments. These environments compute audio in real-time, with data from input devices and UI elements, and may, in addition, display graphical or video feedback.

On-the-fly environments possess elements of both programming and runtime systems – and most importantly, the runtime capability to modify the structure and logic of the executing program itself. Several existing environments possess varying degrees of on-the-fly capabilities. Max and Pd give programmers limited ability to change their patches at runtime. The SuperCollider programming environment allows for synthesis patches to be sent and added to a server in real-time. Another interesting system for runtime graphical and virtual-reality programming is Alice (Pausch et. al. 1995), which allows users to create a virtual world, and to add and modify behaviors using a high-level scripting language (Python in this case) on-the-fly. This rapid-prototyping graphical environment is notable for having no hard distinction between development and runtime. Similarly, MATLAB (Mathworks), while not intended as a real-time programming tool, has a command line that directly uses statements from the language and embodies a similar immediate run aesthetic. However, many existing environments lack a unified timing framework.

3. ChuckK + On-the-fly Programming

The Audicle is based on our previous work with the ChuckK synthesis language and *on-the-fly programming*. ChuckK’s *timed concurrency* model, on-the-fly compiler, and our operational semantics for on-the-fly programming help form the foundation of the Audicle. In this section, we overview the features of ChuckK and the semantics of on-the-fly programming that are pertinent to the Audicle.

3.1 ChuckK

ChuckK is designed to be a concurrent, on-the-fly audio programming language (Wang and Cook, 2003). It is not based on a single existing language but is built from the ground up. It is *strongly-typed* and *strongly-timed*, and runs in a virtual machine with a native audio engine and a user-level scheduler. The following ChuckK features and properties are pertinent to the Audicle:

- A straightforward way to connect data-flow / unit generators.
- A sample-synchronous timing mechanism that provides a consistent, unified view of time, embedded directly in the program flow. Time is fundamentally decoupled from data-flow.
- A cooperative multi-tasking, concurrent programming model *based on time* that allows programmers to add concurrency easily and scalably. Synchronization is accurately and automatically derived from the timing mechanism.
- Multiple, simultaneous, arbitrary, and dynamically programmable control rates via timing and concurrency.
- A compiler and virtual machine that run in the same process, both accessible from within the language.

ChuckK's programming model addresses several problems in computer music programming: representation, level of control for data-flow and time, and concurrency. We summarize the features and properties of ChuckK in the context of these areas. In doing so, we lay the foundation for describing the semantics of on-the-fly programming and the Audicle.

Representation. At the heart of the syntax is the ChuckK operator: a group of related operators (\Rightarrow , \rightarrow) that denote interconnection and direction of data flow and control flow. A unit generator (ugen) patch can be quickly and clearly constructed by using \Rightarrow to connect ugen's in a left-to-right manner (Figure 2). Parameters to the unit generators can be modified using the *single* ChuckK operator, \rightarrow (in conjunction with the timing mechanism). The *double* ChuckK operator, \Rightarrow , by default, deals only with *data-flow*, leaving the issue of *time* to the ChuckK timing mechanism.

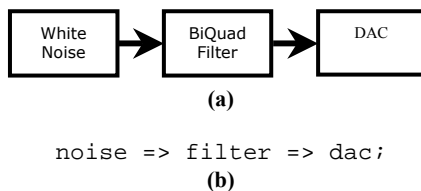


Figure 2. (a) A noise-filter patch using three unit generators. (b) ChuckK statement representing the patch. `dac` is the global sound output variable.

Levels of Control. In the context of audio programming, we are concerned not only with control over *data* but also over *time*. The latter deals with control and audio rates, and the manner in which time is reasoned about in the language. In ChuckK, the key to having a flexible level of control lies in the timing mechanism, which consists of two parts. First, time (`time`) and duration (`dur`) are native types in the language. Time refers to a point in time, whereas duration is a finite amount of time. Basic duration variables are provided by default: `samp` (the duration between successive samples), `ms`

(`millisecond`), `second`, `minute`, `hour`, `day`, and `week`. Additional durations can be inductively constructed by performing arithmetic on existing time and duration values.

Secondly, there is a special keyword `now` (of type `time`) that holds the current ChuckK time, which starts from 0 (at the beginning of the program execution). `now` is the key to reasoning about and manipulating time in ChuckK. Programs can read the globally consistent ChuckK time by reading the value of `now`. Also, assigning time values or adding duration values to `now` causes time to *advance*. As an important *side-effect*, this operation causes the current process to *block* (allowing audio to compute) until `now` actually reaches the desired point in time. (Figure 3)

```
// construct a unit generator patch
noise  $\Rightarrow$  biquad  $\Rightarrow$  dac;

// loop: update biquad every 100 ms
while( true )
{
    // sweep biquad center frequency
    500 + 300 * sin(now*FC)  $\rightarrow$  biquad.freq;

    // advance time by 100 ms
    100::ms  $\Rightarrow$  now;
}
```

Figure 3. A control loop. The \rightarrow ChuckK operator is used to change a biquad's frequency control parameter. The last line of the loop causes time to *advance* by 100 milliseconds – this can be thought of as the control rate.

The mechanism provides a consistent, sample-synchronous view of time and embeds timing control directly in the code. This strong correspondence between timing and code makes programs easier to write and maintain. *Data-flow* is decoupled from *time*. Furthermore, the timing mechanism allows for control rates to be fully throttled by the programmer – audio rates, control rates, and high-level musical timing are unified under the same mechanism.

Concurrent Audio Programming. The intuitive goal of concurrent audio programming is straightforward: to write concurrent code that shares data as well as time (Figure 4).

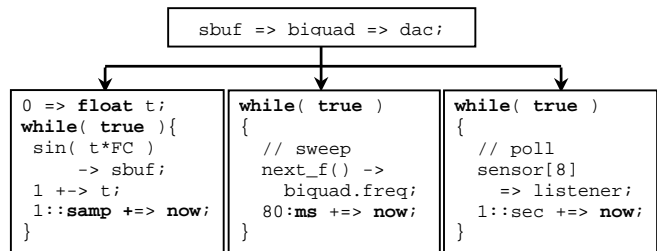


Figure 4. A unit generator patch and three concurrent paths of execution at different control rates.

ChuckK introduced the concepts of *shreds* and the *shreduler*. A shred is a concurrent entity like a thread. But unlike threads, a shred is a deterministic shred of computation, synchronized by time. Each of the concurrent paths of execution in Figure 4 can be realized by a shred. They can reside in separate source files or be dynamically spawned (*sporked*) from a single parent shred.

The key insight to understanding concurrency in ChuckK is that shreds are automatically synchronized by time. Two independent shreds can execute with precise timing relative to each other and to the virtual machine, without any knowledge of each other. This is a powerful mechanism for specifying and reasoning about time locally and globally. Furthermore, it allows for different control rates to execute concurrently and accurately. ChuckK concurrency is orthogonal in that programmers can add concurrency without modification to existing code. It is also scalable, because shreds are implemented as efficient user-level constructs in the ChuckK Virtual Machine.

3.2 On-the-fly Programming

On-the-fly programming is the practice of adding or modifying code at runtime and presents several challenges that include modularity, timing, manageability, and flexibility (Wang and Cook, 2004). Our previous work described a formal on-the-fly programming model based on the timing and concurrency of ChuckK and included both an *external* and an *internal* semantic, as well as an *on-the-fly performance aesthetic* (Figure 5). The external interface uses shreds to provide modularity. The internal semantic provides a means to precisely synchronize the shreds using the timing mechanism. The Audicle both leverages this model and facilitates its usage.

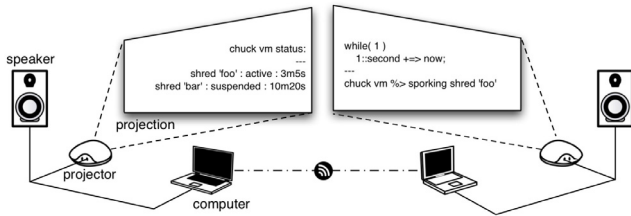


Figure 5. An instance of the on-the-fly performance aesthetic. Each computer is projected at runtime, allowing for the programming process to be visually communicated to the audience, constructing a live correspondence of the intentions to the results.

External Interface. The on-the-fly programming model, at the high-level, can be described in the following way. A ChuckK virtual machine begins execution, generating samples (as necessary), keeping time, and waiting for incoming shreds. New shreds can be *assimilated* on-the-fly into the virtual machine, sharing the memory address space, the global timing mechanism, and is said to be *active*. Similarly, an active shred can be *dissimilated*, or removed from the virtual machine, or it can be suspended or be replaced by another shred. This interface is designed to be simple, and delegates the actual timing and synchronization logic to the code within the shred (discussed in the next section).

The high level commands to the external interface are listed below. They can be invoked on the command-line, in ChuckK programs (as functions calls to the `machine` and `compiler` objects), over the network, or – as we will see in Section 4 – using the Audicle.

- *Execute* – starts a new instance of the virtual machine in a new address space.
- *Add* – type-checks, compiles, and *sporks* a new shred (from ChuckK source file, a string containing ChuckK code, or a pre-compiled shred).
- *Remove* – removes a shred by ID from the virtual machine.
- *Suspend* – suspends and places a shred on the *shredulers*'s suspended list.
- *Resume* – resumes a suspended shred. The shred will begin execution at the suspended point in the code.
- *Replace* – invokes remove followed by an add.
- *Status* – queries the status of the virtual machine for the following information: (1) a list of all active shreds ID's, source/filename, duration since assimilation (*spork time*), (2) information on virtual machine state: current shredder timeline, CPU usage, synthesis resources.

As an example, Figure 6 shows code that adds a new shred from file to the virtual machine using two different methods.

```
# add foo.ck
shell%> chuck --add foo.ck
```

(a)

```
// compile shred from file "foo.ck"
compiler.compile( "foo.ck" ) => shred foo;
// advance time by 500 milliseconds
500::ms ==> now;
// spork "foo"
machine.add( foo );
```

(b)

Figure 6. Two examples of using the runtime shred management interface: (a) from a command-line shell, (b) from within a shred, which has precise timing control.

Internal Semantics. The internal semantics of our on-the-fly model deal with the problem of precise timing and synchronization between on-the-fly modules – shreds in this case. In our model, the semantics are natural extensions of the ChuckK timing mechanism. By querying and manipulating time using the special variable `now`, the programmer can determine the current time, and specify how the code should respond. By the properties of ChuckK timing and concurrency: (1) `now` always holds the current ChuckK time. (2) Changing the value of `now` *advances* time and has the side effect of blocking the current shred (allowing audio and other shreds to compute) until `now` “reaches” the desired time. (3) If `t` is of type time, `t ==> now` advances time until `t` equals `now`. (4) If `d` is a duration, `d ==> now` advances time by `d`. We illustrate this below with some code segments that synchronizes to time.

- Let time pass for some duration (in this case 10 seconds)


```
now + 10::second ==> now;
// or simply:
10::second ==> now;
```
- *Synchronize* to some absolute time `t`

```
t ==> now;
```
- *Synchronize* to some absolute time `t` (or after)


```
if( t < now ) t ==> now;
```
- *Synchronize* to the beginning of next period of duration `T`

```
120::ms ==> dur T; // period to synchronize to
T - (now % T) ==> now; // advance time by remainder
```
- *Synchronize* to the beginning of next period, plus offset `D`

```
T - (now % T) + D ==> now;
```
- Start as soon as possible


```
// no code necessary
```

4. The Audicle

The Audicle is a graphical on-the-fly audio programming environment. It is based on the semantics of on-the-fly programming and ChuckK’s *strongly-timed* concurrency model. This powerful feature of the language is extended in the Audicle in that concurrency is also *visualized*. Thus, the Audicle’s graphical aesthetic is given significant consideration in the design: it is to be visually bold, colorful, and minimal. It is to provide an orthogonal set of tools and visualizations that can be combined into more complex configurations and usages. Much of the information is conveyed by simple 3D shapes and structures, which can be viewed at high frame-rates, from virtually any angle or distance. The Audicle is implemented *exclusively* using 3D graphics, running in full-screen or windowed mode. We present the design and properties (Sections 4.1 and 4.2) and discuss its implementation (Section 4.3). We show that the Audicle is capable of achieving expressive, high-quality synthesis in combination with high-performance visualizations.

4.1 Design of the Audicle

In the Audicle, there is no distinction between development and runtime: *all* components are fully accessible at runtime. This integration is based on the ChuckK compiler and virtual machine, augmented with a smart editor and interfaces for viewing and manipulating concurrency, timing, and system usage/connectivity, etc. The design philosophy is one of runtime cohesion of phases and visualizations of system state.

There are also several important instances of *decoupling* in the Audicle. As in ChuckK, *data-flow* and *time* are fundamentally decoupled, leading to more expressive and clearer audio programs. The Audicle is based on a decoupled simulation model for virtual reality (Shaw et. al, 1992). In this model, the simulation can operate at an arbitrary rate *independent* of the graphics rendering-rate, leading to smoother graphics and more flexibility in the simulation algorithms. In the Audicle, audio synthesis, graphics, and simulation are *loosely-coupled*, with the highest priority given to audio computations and the virtual machine.

Out of the desire to provide a simple, “graspable” virtual environment and interface, we associate the Audicle with a simple geometric shape. The various parts of the Audicle are mapped and displayed on the faces of a *virtual cube*, called the **Audicube** (Figure 7a). At any time, the user can interact with one face, and have the ability to move to others faces by using hotkeys (USER_KEY+[up|down|left|right|face#]), graphical interface, Audicle shell commands (`%> face 4 -OF- %> face shredder`), or even ChuckK statements (`audicle.face(4);`).

There is a slim command-line console (Figure 7b) that can be invoked to appear over the currently active face. The console resembles that of MATLAB, where statements of the underlying language can be entered directly on the command-line. The key difference is that the Audicle console also has a *strong notion of time*. For example, it is possible to write a console command that loops and fires off *N* number of shreds, 100ms apart (see example in Figure 7b). Indeed, the Audicle console *itself* is a powerful on-the-fly programming tool.

On the Audicube, there are 5 primary faces plus one blank *tabula rasa* face for real-time graphics or for use as a visual scratchpad. They are listed below and discussed in detail in Section 4.2.

- *Concurrent Editor* – “smart” editor interface
- *Compiler-space* – deep structure of the program
- *VM-space* – system resource and I/O management
- *The Shredder* – shred and concurrency management
- *Time and Timing (TnT)* – time management
- *Tabula Rasa* – blank slate (“anything goes here”)

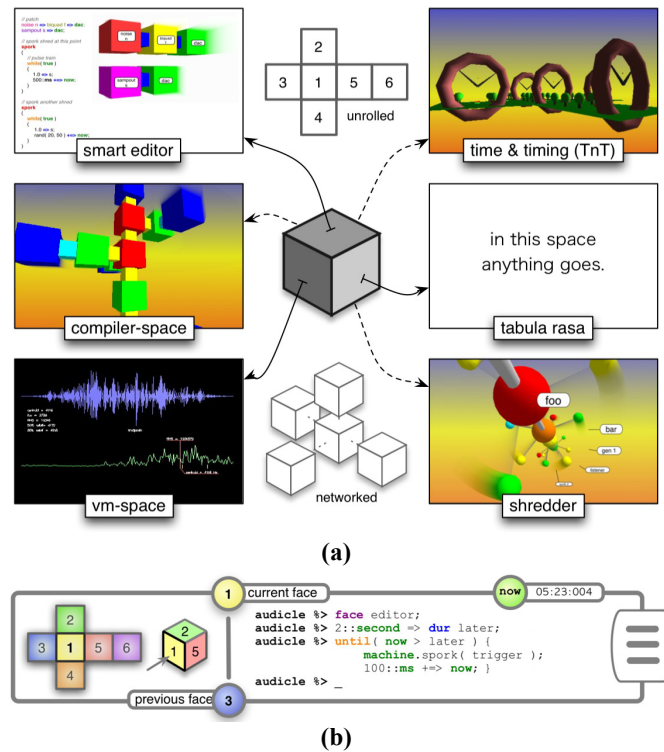


Figure 7. (a) Faces of the Audicube. The ‘cube can be “unrolled”, or “stacked” into a networked configuration. The six faces are shown on the two sides. **(b) Audicle Console.** The interface (left) can be used to graphically navigate the Audicube. The on-the-fly command prompt (right) accepts ChuckK statements and Audicle commands (built using ChuckK macros).

Additionally, the Audicle is designed to be a networked, collaborative development/performance environment. The simplest type of collaboration involves several *Audiclae* connected over a network sharing a central virtual machine for audio synthesis and computation. In this scheme, precise timing can be preserved. For example, each Audicle can send pieces of ChuckK code over TCP/IP to be compiled and executed on a central Audicle, with precise timing embedded within the code. In a more complex topology, multiple Audiclae can each have access to a virtual machine or perform load-balancing for computation tasks. This offers more overall computational power, but requires timing and synchronization that is beyond the default timing mechanism. We leave this more complex scheme to be investigated in future work.

4.2 Faces of the Audicube

Concurrent Editor. The editor is the primary interface to enter/edit code (an alternative is the console). It is *on-the-fly* – new code can be written and *assimilated* while existing code is running in the virtual machine. The editor is also *concurrent*, allowing programmers to create both serial and parallel code blocks by constructing a *flow-graph of code* (Figure 9a). A new concurrent shred can be introduced by adding a code block. Directed edges in the graph determine the parent/child relationship between concurrent code blocks. Furthermore, blocks can also be named and reused.

There is an one-to-one correspondence between concurrent ChucK code and the flow graph – the edges in the graph map to *spork* statements in ChucK, allowing for any ChucK program to be uniquely represented as code or as a graph. Also, the editor automatically visualizes ChucK statements using simple objects (Figure 9b). The result is a 3-dimensional, Max/Pd-like sub-environment with a very important underlying difference: the “smart” editor only displays *data-flow*, leaving *time* to be dealt with separately in the code using ChucK’s timing mechanism, again decoupling data and time.

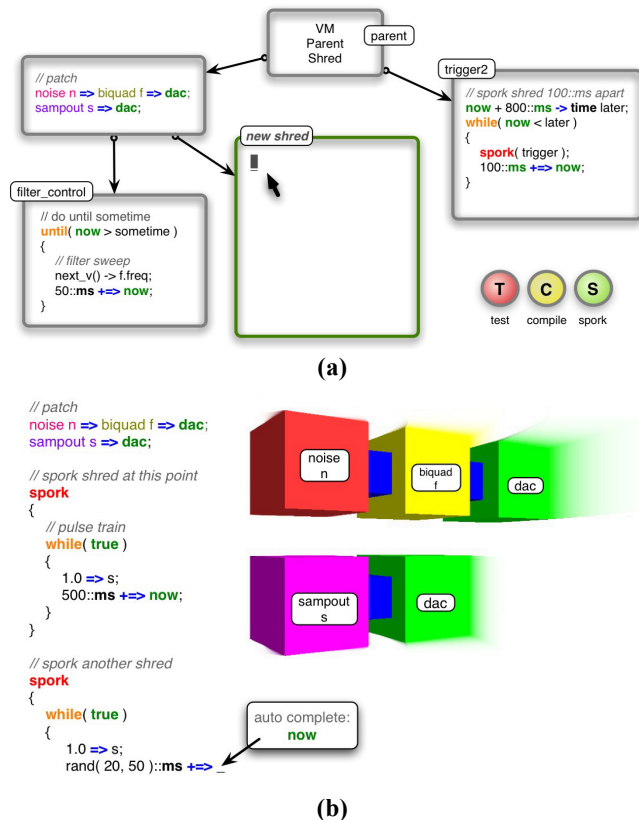


Figure 8. Concurrent Editor. (a) The editor allows the programmer to construct *flow-graphs* of concurrent code. Arrows represent *sporking* of new *shreds*. Code can be on-the-fly edited, tested, compiled, and *sporked* using the editor or the console. (b) ChucK code is entered (left). In the same space, parts of the program structure are visualized. In this example, the two simple patches (left top) are automatically represented by similarly-colored objects (right).

The editor is “smart” in several sense. First, it has access to the deep structure of the existing ChucK program, including type and timing information, in addition to results gathered from live background processing. The editor can use this information to suggest potential statement completions (class members, type, etc.). This is similar to features in existing commercial integrated development environments (IDE’s) such as Microsoft Visual Studio. Also, the editor serves an important ChucK-specific task – that of assisting resolutions of the massively-overloaded ChucK operator.

Second, since ChucK programs are type-checked and emitted into virtual ChucK instructions, the editor can serve as an “on-the-fly debugger”. It is possible to halt the program at any instruction and change the code before or after it.

Finally, there are various options for running and testing the code. These can be accessed via the graphical interface buttons, keyboard shortcuts, or through the Audicle console. One useful option is to “test-run” the code in the current VM in a protected “sandbox” environment.

Compiler-space. The compiler-space face of the Audicle allows viewing of the deep structure of the program gathered during compilation. Many aspects of the program structure can be visualized here. The syntax tree, with typing annotations is available for all code segments. One can see the ChucK virtual instructions emitted form a piece of code. Also, the user can examine the values of global values, or select a shred (from the console or the Shredder) and examine values of local variables during runtime. It is possible to traverse the compiler-space in 3D and to select code segments to be displayed in the editor.

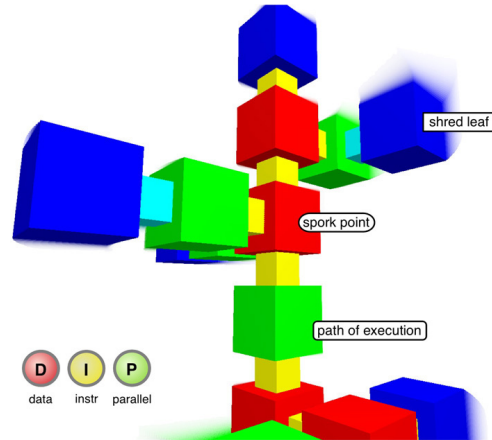
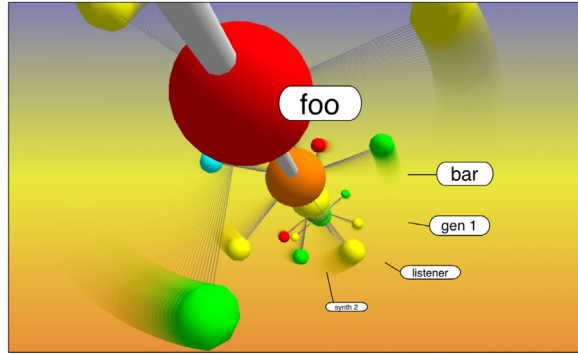


Figure 9. Compiler-space Explorer. In this example, the real-time *shred-sporking* pattern for a code segment is shown, with red objects representing points in the code where shreds were *sporked*, and blue and green representing paths of execution. The structure grows dynamically. Clicking on the objects invokes appropriate actions (viewing code or switching to another face).

The Shredder. The Shredder visualizes and manages shreds and concurrency. While it is possible to do this from within the language or via the console, the Shredder provides a more straightforward (and often more timely) interface to add/remove/modify shreds. The Shredder operations can also be mapped to input devices, allowing shreds to be managed with minimal typing and other user-introduced latency.

Programmers can view all the shreds in the virtual machine using a text-based list and/or a graphical tree-view (Figure 10). Because code can generate code or originate from other *Audiclae*, the shreds in the Shredder are identified as local, generated, or remote. Additionally, shred id/name, creation time, number of cycles computed, number of context switches, and state (active, suspended) are available for each shred.



(a)

	id	state	name	owner	origin	spork time	cycles	context	avg. ctrl rate
●	101	[active]	[foo]	[me]	[file:foo.ck]	[00:00]	[8.5M]	[8456]	[35::ms]
●	102	[wait]	[gen 1]	[me]	[editor]	[00:30]	[7.4M]	[1032]	[290::ms]
●	105	[wait]	[analysis]	[remote]	[audicle:3]	[00:56]	[5M]	[802]	[274::ms]
●	108	[suspend]	[bar]	[me]	[editor]	[02:01]	[4.3M]	[4421]	[68::ms]
●	109	[wait]	[poll 1]	[me]	[file:poll.ck]	[01:10]	[200K]	[2034]	[147::ms]
●	110	[wait]	[poll 2]	[me]	[file:poll.ck]	[03:48]	[54K]	[300]	[1::second]
●	200	[wait2]	[synth 2]	[me]	[shredder]	[04:31]	[3.2M]	[2034]	[147::ms]
●	303	[wait2]	[listener]	[AUD]	[audicle]	[00:00]	[10K]	[10242]	[29::ms]
●	120	[wait]	[renderer]	[AUD]	[audicle]	[00:00]	[800K]	[210]	[1.4::second]
●	300	[wait2]	[daemon]	[VM]	[vm]	[00:00]	[22K]	[9035]	[33::ms]

(b)

Figure 10. *The Shredder.* (a) Visualization of shred activity, the larger spheres are parent shreds. The smaller child shreds “orbit” the parents at a rate proportional to their current average control rate. The names refer to the parents. (b) On-the-fly timetable of shreds in the virtual machine.

An important use of this mechanism is to monitor and manage currently executing shreds, and to identify *hanging* or *non-cooperative* shreds. For example, if the system runs a shred containing an infinite loop, it will fail to yield and cause the virtual machine execution unit to hang indefinitely. This type of behavior cannot be reliably detected at compile time due to the Halting Problem. However, the on-the-fly programmer can identify and remove misbehaving shreds from the virtual machine manually by suspending them in the Shredder (and potentially debugging them immediately in the editor), resulting in minimal interruption to the session. While this recovery mechanism is far from perfect, it is far more advantageous over killing the system and restarting. Similarly, it can help the programmer optimize the system by identifying shreds that are consuming too much CPU time.

Time and Timing (TnT). This face is a visualization of the *shreduling* at runtime. It is like an *electrocardiogram (EKG)*. Several parallel lines, representing active shreds, move along in

Chuck system time. A spike is encountered when a shred executes at a particular point in time. The nature of the time-based concurrency in Chuck implies that shreds only executes at discrete *points* in time, and must explicitly allow time to advance. So the spikes represents the relative frequency that each shred is currently *shreduling* itself. The height of the spike is mapped to some measure of physical resources (such as compute cycles consumed). (Figure 11)

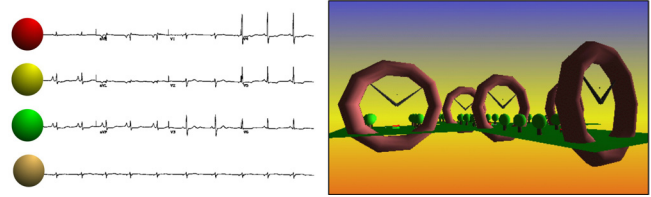


Figure 11. (left) *EKG of the shreds.* Spike height is a quick estimate of computations per execution in time. (right) *Clock array visualization* of average shred control rates.

This visualization presents a way to gain a high-level glance at the overall timing behavior. For example, it is easy to see how frequently each part of the system is computing and gain a rough idea of resources used as a function of time.

VM-space. The VM-space contains useful (or interesting) runtime information and statistics about the Audicle and virtual machine. It is responsible for management of input devices, interface protocols (MIDI, OSC, SKINI), dynamic linking, and network connectivity (including establishing connections with remote Audiclae). Additionally, CPU load, unit generator resources, and analysis output (such as FFT and audio feature extraction), can be displayed here.

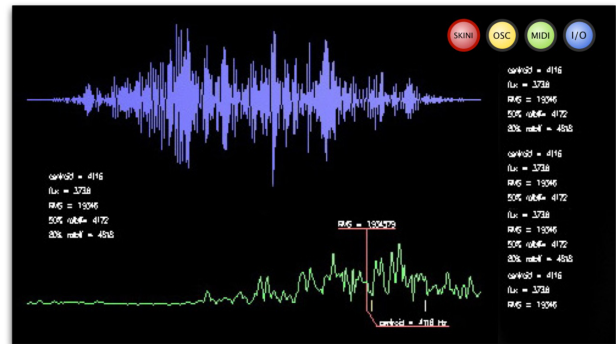


Figure 12. *The VM-space Explorer.* In this case, we see the input waveform (top-left) and its FFT magnitude spectrum (bottom-left), as well as resource usage (right).

Tabula Rasa. The “blank slate” face is actually not limited to one face, but any number of faces as requested by the programmer. Its content is fully programmable using Chuck and the OpenGL API. In addition, the Chuck graphical programming interface includes access to the Audicle graphics and windowing engines, along with its set of minimal user interface widgets (used in the rest of the Audicle). This space makes the Audicle suitable for rapidly developing custom, high-performance graphical front-ends for synthesis, analysis, and performance.

4.3 Implementation

The Audicle's implementation consists of a graphical rendering engine, a low-latency I/O & networking framework, a minimal windowing system, in addition to an augmented ChuckK virtual machine. The implementation is in C/C++, (some higher-level components are written using ChuckK) and reuses many data structures from ChuckK's compiler and virtual machine. All components run in the same address space.

The graphics-rendering engine of the Audicle uses the OpenGL API and runs on Mac OS X, Linux, and Windows. With recent hardware advancements, using 3-D graphics (exclusively) with real-time synthesis is highly feasible. (1) With even modest graphics hardware support, the vast majority of the rendering can take place on the GPU (graphics processing unit), leaving CPU cycles (85-90%) for synthesis. (2) Using custom-built, minimal user interface elements, we can handle user-interface events more efficiently than the windowing sub-system, and with potentially better responsiveness. Because the rendering-rate stays relatively constant (at ≥ 30 frame/second), the CPU usage stays constant and is less subject to large bursts due to user interface processing. (3) 3D graphics is flexible. It can "emulate" 2D when needed, and also provides significant viewing freedom.

5. Conclusions and Future Work

On-the-fly programming opens the potential for interesting interactions and visualizations in the *audio programming process*. Through the different faces in the Audicube, the programmer, composer, and performer can develop code in a truly concurrent editor, then visualize its behavior in terms of concurrency, timing, and as well as the runtime state of the system. Concurrency, a natural and useful way to represent many concepts in sound and music, is captured by ChuckK, and visualized by the Audicle.

The integrated, on-the-fly environment of the Audicle completes the development-to-runtime loop. The result is greater than the sum of its parts. The expressive power of coding is made available for runtime manipulation. In turn, on-the-fly information from *runtime* aids the *development* process, expanding the horizons of both. We gain the advantages of immediate feedback in an always-modifiable continuum.

Additionally, the Audicle motivates a new kind of *audio programming mentality* – one involving continuous exploration and experimentation. Recall on-the-fly programming from Section 3.2. The Audicle further motivates its notion of runtime programmability as a new form of *performance aesthetic*, where code is used to expressively control the synthesis and the process is conveyed to the audience. It also provides a platform where a degree of *virtuosity* can evolve. Due to its visual nature and immediate feedback, the Audicle can be a useful compositional environment, where the composer can incrementally work on concurrent parts of a program piece. Similarly, it could function as an educational tool, for teaching synthesis, audio programming, or multimedia.

Potentially, the Audicle is the beginning to a new class of environments for developing synthesis programs on-the-fly, as well as for visualizing the audio programming process. It

would be interesting to further investigate new interfaces for on-the-fly editing and code control, and new types of visualizations. Also, future work can investigate the technical and aesthetic aspects of collaborations between remotely connected Audiclaes, as well as devise new on-the-fly programming systems and environments. We believe these potentials are worth exploring.

<http://chuck.cs.princeton.edu/>

<http://on-the-fly.cs.princeton.edu/>

6. Acknowledgements

We sincerely thank Ari Lazier, Nick Collins, Alex McLean, Adrian Ward, Julian Rohrerhuber, and the Princeton Graphics Group for their encouragement, ideas, and support.

References

- Dannenberg, R. B. and E. Brandt. 1996. "A Flexible RealTime Software Synthesis System." In *Proceedings of the International Computer Music Conference*. International Computer Music Association, pp. 270-273.
- Dannenberg, R. B. 1997. "Machine Tongues XIX: Nyquist, a Language for Composition and Sound Synthesis." *Computer Music Journal* 21(3):50-60.
- Lee, E. 2003. "Overview of the Ptolemy Project," *Technical Memorandum UCB/ERL M03/25*, University of California, Berkeley.
- McCartney, J. 1996. "SuperCollider: A New Real-time Synthesis Language." In *Proceedings of the International Computer Music Conference*. International Computer Music Association.
- Mathworks, Inc. *MATLAB Documentation*.
<http://www.mathworks.com/>
- Pausch, R., T. Burnette, A. C. Capeheart, M. Conway, D. Cosgrove, R. DeLine, J. Durbin, R. Gossweiler, S. Koga, J. White. 1995. "Alice: Rapid Prototyping System for Virtual Reality." In *IEEE Computer Graphics and Applications*, May 1995.
- Puckette, M. 1991. "Combining Event and Signal Processing in the MAX Graphical Programming Environment." *Computer Music Journal* 15(3):68-77.
- Puckette, M. 1997. "Pure Data." In *Proceedings of the International Computer Music Conference*. International Computer Music Association, pp. 269-272.
- Schubiger S. and S. Muller. 2003. "Soundium 2: An Interactive Multimedia Playground." In *Proceedings of the International Computer Music Conference*. International Computer Music Association, pp. 301-304.
- Shaw, C., J. Liang, M. Green, and Y. Sun. 1992. "The Decoupled Simulation Model for Virtual Reality Systems." In *Proceedings of the ACM SIGCHI Human Factors in Computer Systems Conference*. pp. 321-328.
- Vercoe, B. and D. Ellis. 1990. "Real-Time CSOUND: Software Synthesis with Sensing and Control." In *Proceedings of the International Computer Music Conference*. International Computer Music Association, pp. 209-211.
- Wang, G. and P. R. Cook. 2003. "ChuckK: a Concurrent and On-the-fly Audio Programming Language." In *Proceedings of the International Computer Music Conference*. International Computer Music Association, pp. 219-226.
- Wang, G. and P. R. Cook. 2004. "On-the-fly Programming: Using Code as an Expressive Musical Instrument." In *Proceedings of the International Conference on New Interfaces for Musical Expression*. (To be published).