CO-AUDICLE: A COLLABORATIVE AUDIO PROGRAMMING SPACE

Ge Wang   *Ananya Misra*   *Philip Davidson*   *Perry R. Cook*[†]
Princeton University
Department of Computer Science ([†]also Music)

ABSTRACT

The Co-Audicle describes the next phase of the Audicle's development. We extend the Audicle to create a collaborative, multi-user interaction space based around the ChucK language. The Co-Audicle operates either in client/server mode or as part of a peer-to-peer network. We also describe new graphical and GUI-building functionalities. We draw inspiration from both live interaction software, as well as online gaming environments.

1. INTRODUCTION AND MOTIVATION

Co-Audicle describes the next phase of the Audicle's development. We extend the environment described in the Audicle [8] to create a collaborative, multi-user interaction space based around the ChucK language [6]. The Co-Audicle operates either in client/server mode or as part of a peer-to-peer network. We also describe new graphical and GUI-building functionalities. We draw inspiration from both live interaction software, as well as online gaming environments.

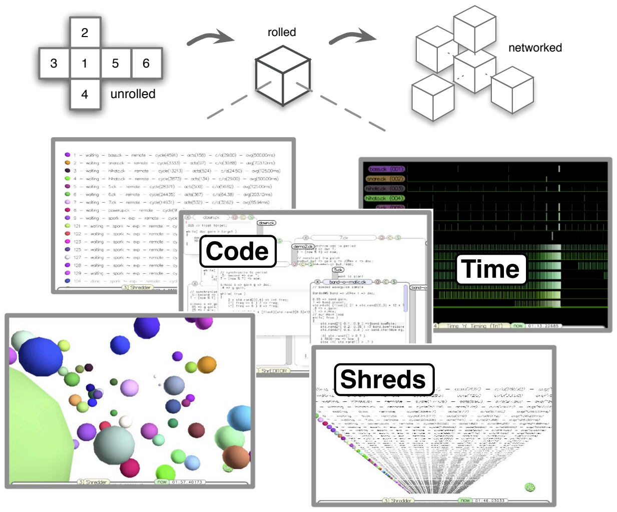

Figure 1. Faces of the Audicle. As shown on the top, the Audicle is meant to be networked.

On-the-fly programming [7] sees code as an expressive musical instrument and the act of programming as performance. This framework and today's fast networks provide a unique opportunity to extend this practice to let many people, across a wide geography, build and play one instrument. This is our primary motivation.

There have been various research projects looking at aspects of this type of collaboration. [2, 5, 1]. The Co-Audicle's focus is code and the act of programming audio in a real-time, distributed environment. It is concerned with the topology of the collaboration and the levels of guarantee about timing, synchronization, and interaction associated with each.

The ChucK/Audicle framework provides a good platform and starting point for the Co-Audicle. ChucK programs are strongly-timed, which means they can move to a new operating environment on another host, and find out and manipulate time appropriately. ChucK programs are a concise way to describe sound synthesis very precisely. The latter is useful in that many sounds or passages can be transmitted in place of actual audio, greatly reducing need for sustained bandwidth. The Audicle graphical programming environment visualizes the timing and processing in the programs and virtual machine, and already has a context-sensitive editor for managing and creating ChucK programs. These components are a good foundation for the Co-Audicle.

2. MODEL 1: CLIENT / SERVER

In the client/server model, multiple client Co-Audicles connect to a centralized Co-Audicle server. All computations, including audio synthesis, happen at the server. The clients act as "dumb" terminals that send code and control requests to the server and also receive the synthesized audio streams from the server. The server also returns statistics and metadata with each frame of audio, such that the VM state and shred timing can be visualized at the client Co-Audicles. Furthermore, multiple clients can collaboratively edit the same code module.

2.1. Co-Audicle Server

A Co-Audicle server can be instantiated by any user. The person who started the server is by default the superuser and moderator. Upon start-up, the moderator can also set rules for the server, such as the maximum number of clients, and security modes (such as disallowing certain operations from shreds).

Once the server is running, clients can join. The client can learn of the server's existence and address in one of several ways. The parties involved can agree beforehand on the server host and connect to it by IP. It is also possible to automatically discover servers on the local area network. Additionally, servers can optionally register themselves with an internet-wide directory.

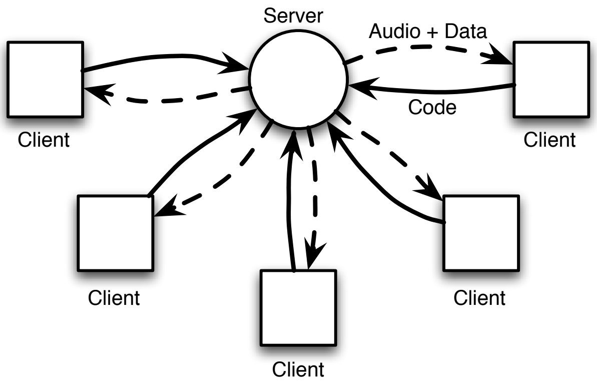

Figure 2. Co-Audicle client/server model.

The server contains a ChucK virtual machine that runs all the shreds and synthesizes audio. It also manages the list of clients, as well as a list of active ”edit rooms”, where collaborative editing can take place. As the audio is synthesized, it is broadcast back to the clients, but also with the statistics of shred timing, and VM processes. The statistics and other meta-data are associated with the audio using ChucK timestamps. This allows the client Co-Audicles to visualize the virtual machine as if it were running locally.

The server is a space where clients can move around and discover other clients. By default, each client starts in his/her own edit room. Clients can leave their own edit room and move to other rooms to collaboratively edit code (if the room owner approves) or to observe the editing in real-time. Feedback can take place through separate ’chat’ windows or a real-time ”annotation” function where a segment of code may be flagged with a comment from another user.

2.2. Co-Audicle Client

The Co-Audicle client behaves as a dumb terminal that does not perform VM computations or audio synthesis. The client allows the programmer to edit code on the server. It also receives data to be visualized, such as shred activity, as well as synthesized audio from the server. The client plays the audio synchronized with the visualization, giving the impression that everything is happening locally. The visualization at the client involves the totality of shreds and timing at the server. Each client can observe other users’ shreds and timing, or filter out the ones he/she does not want to see.

In addition to editing commands, the client also sends out of band messages for connection, user chat, and navigation of the server space.

3. MODEL 2: PEER-TO-PEER

The Co-Audicle can also operate under a peer-to-peer model. Under this scheme, each Co-Audicle forms a node, and every node runs a ChucK virtual machine and synthesizes audio. The main idea in this model is that no audio is transferred among the nodes, only code and meta-data. ChucK code is directly mapped to time and synthesis, and serves as a convenient and compact alternative to sending audio. This allows the system to scale to hundreds or more nodes that only need to replicate code. Of course, there are many challenges to this:

Data consistency. Consistency across nodes is maintained in two ways. Since ChucK programs can find out about time and act accordingly, the programs themselves can adapt upon arriving at a new node. In this sense, the same code behaves intelligently on different nodes. Also there is a notion of a multi-environment shred syndicate (or a *mess*) that can be internally sophisticated but does not access data outside the *mess*, although it can access time. A *mess* could be replicated across nodes without worrying about data-dependency. In this way, a particular sound or passage could migrate from node to node, and share only timing with the rest of the system, with no side effects on states.

Synchronization. *Messes* can be synchronized via the timing mechanism in ChucK. Shreds in a *mess* can query for the current time on the VM and act appropriately. However, a bigger problem is clock skew among all the nodes. The ChucK virtual machine is data-driven by the number of samples it computes and some soundcards may clock slightly faster than others. In the long run, time-consistency is hard to maintain across the system. In the Co-Audicle, a global clock resynchronization propagates periodically, paired with dynamic *mess*-level adjustments.

Security. The owner of each node can determine the level of security for his/her node. A high security level can mean denying access to operations outside of the virtual machine. The mechanism is like the one used in Java applets.

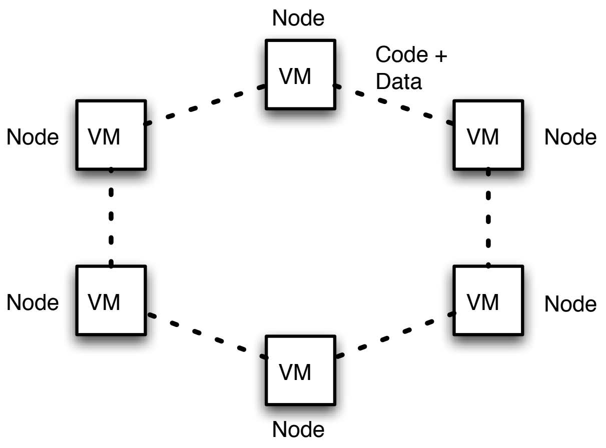

Figure 3. Co-Audicle peer to peer model. No audio is transferred - only code and data.

4. CHUI : CHUCK USER INTERFACE

Since the Audicle is a graphical environment, we also include the ability to program real-time graphics and build GUIs as part of the collaboration. The ChucK language allows for programmatic control interaction, but in many applications we also desire the ability to use a graphical interface, or to provide a user-friendly ”front end” for the performer. CHUI serves as a framework for dynamically building user interfaces in code to allow rapid design and experimentation. UI elements are represented at the same object level as audio ugens within ChucK code. By using a UI object in place of a standard variable (for example, an int or float) we retain the flexibility of programmable control in-code, but can switch to a graphic representation for visual control as desired.

4.1. UI elements

CHUI implements an extensible set of interface elements.

- **buttons** trigger a specified function
- **switches** and toggles select or switch between objects and modes
- **sliders** standard scalable slider implementations
- **option box** select from a preset list of options
- **display** waveform or spectrogram output from a particular ugen
- **meters** volume display
- **group** group a section of elements for positioning and layout
- **memory** preserve the state of elements or groups.

4.2. Behaviors

Adaptive Rendering methods: UI objects present a standardized interface to the ChucK environment, while allowing their own display or interaction code to be extended through subclassing. CHUI objects could conceivably run within a variety of UI systems (TCL/tk, GLUT, SDL, etc) while presenting the same interface to code

UI elements are aware of whether the interface is currently active. If desired, user interactions can be set to ”override” programmable control through a simple toggle.

Object positioning and grouping may be modified programmatically, or through the UI interface. UI elements register with a ”memory” UI object to enable a particular interface element to preserve UI state in successive execution sessions. Positioning properties are managed by object name to simplify re-mapping and code alterations.

5. GLUCK : VISUAL TOOLKIT

ChucK’s precise timing mechanisms provide an ideal integration of audio and visual elements for live multimedia programming. GLucK provides OpenGL, GLU, and GLUT functionality to shreds running in the ChucK VM. In the Audicle, GLucK manages the interface to the *Tabula Rasa* face, used for spontaneous visual improvisation.

While the UI elements described in CHUI may be embedded within GLucK environments, they are a convenient interface, rather than the sole interaction method. We allow for a wide range of interactive modes for performance.

5.1. Basic Functionality

At its core, GlucK’s most basic use is as a simple wrapper to OpenGL, GLU, and GLUT library functions. These libraries are dynamically loaded per shred, and once context is established, users can call standard OpenGL and GLU functions in synchrony with their audio code.

5.2. Core Use

Beyond basic GL functions, GLucK provides simplified windowing calls for creating and managing separate windows, and remaps the GLUT input functions to fit ChucK’s event model.

5.3. High Level Function

We hope to extend GLucK to serve as a full ’vgen’ based scene graph system, modeled on the same calling expressions used for audio chains. We will also extend support for manipulation of both stored and live video input as a media source and for vision-based control methods.

6. AUDIO CHAIN ANALYSIS

The ChucK VM manages a large number of shreds and objects, many of which exchange data and audio through objects in shared memory. The audio chain analysis provides a view of shred activity through the collection of shared objects that constitute the audio chain.

6.1. Visualization

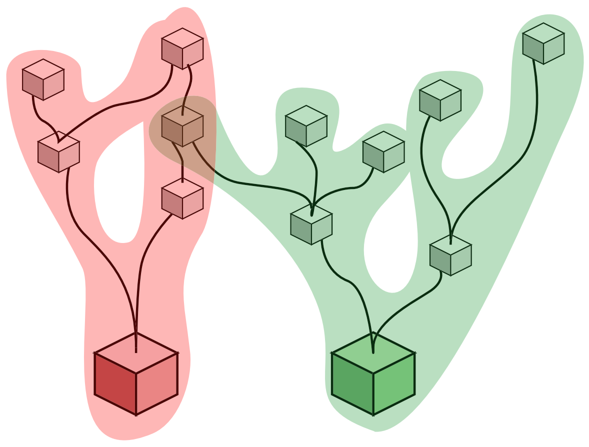

Figure 4. Visualizing shred activity emanating from respective DACs (larger red and green boxes)

To view the structure of the audio processing chain within a particular shred, we dynamically generate a graph similar to those used in Pure Data and other visual programming languages to indicate connections and direction of data flow between unit generators.

For the VM to display activity in multiple shreds, we manage the graph by considering the *aura* that would emanate from each shred's particular DAC, as shown in Figure 4. We generate this *aura* by flagging a particular audio element each time it is accessed through a particular shred. This indication persists according to a decay rate and allows us to render the activity around each element using the assigned *colors* of the respective shreds. Control rates are mapped logarithmically to an intensity scale to indicate rates that could not be displayed or readily perceived by eye.

6.2. Analysis

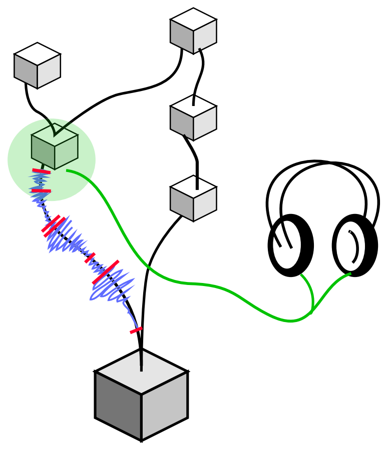

Figure 5. Viewing shred activity(red) and audio output(blue) from a selected ugen element)

We can also view the state of a particular member of an audio chain. In this mode, shown in Figure 5, we allow a user to "peek" at the output of a particular ugen or group of ugens. We use the connections from each ugen outlet to display layers of real-time information, such as ugen activity, waveform, or spectral information.

7. APPLICATIONS

The Co-Audicle can have a wide range of applications in music programming, performance and education. In terms of programming, it enables multiple users to carry out on-the-fly audio programming together from separate workstations. This, in turn, can facilitate massively multi-user distributed live coding, where users are physically spread across the globe but programming different parts of the same system in real-time.

Viewed as a performance, a Co-Audicle concert can be considered a distributed band or orchestra, with each musician performing on his own laptop. On another level, since individual *Co-Audiclae* are closely connected and share code and data, the entire Co-Audicle system can be seen as a single instrument that many performers building and manipulate together. The resulting collaboration patterns, or how each player contributes to the whole, may serve as bases and platforms for experimentation in musical performance.

In addition to synthesizing sound, the Co-Audicle can also be used for collaborative GUI building. This is related to both the performance and programming aspects, as the GUI built in this way can provide the visual aspect of the performance as well as programmatic control.

The distributed nature of the system also supports long-distance music education. For example, the teacher and students can pass code and audio back and forth for live discussions on music and sound synthesis.

8. CONCLUSION

We have presented a preliminary framework for a collaborative audio programming system. The client server and peer-to-peer paradigms are suitable for different types of interactions. This is ongoing work. Our design only addresses a few of the challenges of collaborative audio programming. Many challenges including security and better consistency remain to be resolved.

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

9. REFERENCES

[1] De Campo, A. and J. Rohrhuber. "Waiting and Uncertainty in Computer Music Networks", *Proceedings of the International Computer Music Conference*, 2004.

[2] Jorda, S. Faust Music On Line: (FMOL) An Approach to Real-Time Collective Composition on the Internet, *Leonardo Music Journal 9*, 512, 1999.

[3] McCartney, J. "SuperCollider: A New Real-time Synthesis Language", *Proceedings of the International Computer Music Conference*, 1996.

[4] Puckett, M. Pure Data, *In Proceedings of International Computer Music Conference*. 1996.

[5] Stelkens, J. peerSynth: A P2P Multi-User Software Synthesizer with new techniques for integrating latency in real time collaboration, *Proceedings of the International Computer Music Conference*, 2003.

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

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

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