Contact:
Jason Freeman
Columbia University
Computer Music Center
632 W. 125 Street
New York, NY 10027
jason@music.columbia.edu

AURACLE: A VOICE-CONTROLLED, NETWORKED SOUND INSTRUMENT*

Jason Freeman (Columbia University Computer Music Center), **Kristjan Varnik** (Akademie Schloss Solitude), **Sekhar Ramakrishnan** (Zentrum für Kunst und Neue Medientechnologie), **Max Neuhaus**, **Phil Burk** (Softsynth, Inc.), and **David Birchfield** (Arizona State University, Arts, Media, and Engineering Program)

1. INTRODUCTION

Auracle is a voice-controlled, networked sound instrument conceived by Max Neuhaus and realized collaboratively by the authors. Users interact with each other in real time over the Internet, playing synthesized instruments together in a group "jam." Each instrument is entirely controlled by a user's voice, taking advantage of the sophisticated vocal control which people naturally develop learning to speak.

Auracle was designed to be accessible to a lay public without musical training or technical expertise. We strived to create an open-ended architecture rather than a musical composition: a system which, as much as possible, responds to but does not direct the activities of its users. We also sought to build a highly transparent system, in which users could easily identify their own contributions within an ensemble of participants, which also remained engaging over extended periods of time.

2. HISTORICAL BACKGROUND

2.1 VOICE-CONTROLLED INSTRUMENTS

Auracle uses analysis data from the voice to control a synthesis engine; it does not directly process and output an audio stream. This approach was initially motivated by practical considerations: it reduced network bandwidth and latency while maintaining high quality audio output.

A number of recent software projects and interactive musical works encouraged us to pursue this technique. For example, the Kantos software plugin (Antares 2004) maps pitch, rhythmic, and formant analyses of a monophonic audio input signal onto its synthesizer; parameters of the mappings and the synthesis algorithm itself are specified by the user through a graphical interface. In the Singing Tree (Oliver 1997), a component of the interactive "Mind Forest" in Tod Machover's *Brain Opera* (Machover 1996), users are asked to sing a steady pitch into a microphone; as they hold it steadier and longer, a MIDI harmonization becomes richer, and images on a screen begin to change. And the *Universal Whistling Machine* (Böhlen and Rinker 2004) analyzes the pitch and amplitude envelopes of a user's whistling and synthesizes responses in which the tempo, contour, and direction of the analysis data are transformed.

2.2 SHARED SONIC ENVIRONMENTS

Barbosa defines shared sonic environments as "a new class of emerging applications that explore the Internet's distributed and shared nature [and] are addressed to broad audiences” (Barbosa 2003: 58). As examples, he cites *WebDrum* (Burk 1999), where on-line participants collaboratively alter settings on a drum machine; *MP3Q* (Tanaka 2000), where users collectively manipulate MP3 files with a 3D interface; and *Public Sound Objects* (Barbosa and Kaltenbrunner 2002), an open-ended architecture for the creation of shared sonic environments.

We consider Auracle to be a shared sonic environment, and our work was influenced by Barbosa’s examples as well as other recent projects. In *DaisyPhone* (Bryan-Kinns and Healey 2004), for example, Internet or mobile-phone users collaboratively modify a looping musical MIDI sequence, with each user coloring circles to change the pitches and rhythms in his or her instrument. In *Eternal Music* (Brown 2003), each user drags a ball around a window to control a drone generated by modulated sine waves. And components of both the *Cathedral Project* (Duckworth 2000) and the *Brain Opera* (Machover 1996) have invited Internet users to control sounds during live physical performances, collaborating not only with other Internet users but also with live performers onstage in a concert hall.

But even more than these recent Internet-based environments, we were inspired by analog environments which used telephone and radio networks to create virtual spaces for audio collaboration. Neuhaus’ own Broadcast Works, such as *Public Supply*(1967) and *Radio Net* (1977), used the telephone and radio networks and analog processing modules to alter, mix, and broadcast audio input from callers during live radio performances (Neuhaus 1994). Ongoing radio shows by NegativLand (Joyce 2005) and Press the

Button (Radio Show Calling Tips 2005), among others, enable callers to join improvising musicians in the broadcast studio. And *Silophone* (The User 2000) operates in both the analog and digital domains; it joins together sounds made by telephone callers and soundfiles uploaded by Internet participants, playing them in a giant grain silo in Montreal and broadcasting their acoustic transformations back over the phone and Internet to the participants.

3. ARCHITECTURE

[Figure 1. Auracle System Architecture.]

Users launch Auracle from the project's web site, opening a graphical user interface through which they can "jam" with other users logged in from around the world. To control their instrument, users input vocal gestures into a microphone. Their gestures are analyzed, reduced into control data, and sent to a central server. The server broadcasts that data back to all participating users. Each client computer receives the data and uses it to control a software synthesizer.

The client software is implemented as a Java applet incorporating the JSyn plugin (Burk 1998), and real-time collaboration is handled by a server running TransJam (Burk 2000). Data logging for debugging, usage analysis, and long-term system adaptation is handled by an HTTP post (from Java) on the client side and PHP/MySQL scripts on the server side.

The following subsections describe each architectural component in detail.

3.1 LOW-LEVEL ANALYSIS[1]

The initial low-level analysis of the voice computes basic features of the audio signal over an analysis window. The incoming sound is analyzed for voicedness/ unvoicedness, fundamental frequency, the first two formant frequencies with their respective formant bandwidths, and root mean square (RMS) amplitude.[2] JSyn is used to capture the input from a microphone, but it cannot extract the vocal parameters we need, so we built this functionality ourselves. We limited our own DSP implementation to pure Java to avoid packaging and deploying JNI libraries for each targeted platform. We considered techniques based on linear prediction (LP), cepstrum (used in Oliver 1997), FFT, and zero-crossing counts. We chose linear prediction, feeling it would be the easiest to implement in pure Java with acceptable performance and accuracy.

Raw sample data from the microphone is brought from JSyn into Java. Once in Java, the data is determined to be voiced or unvoiced based on the zero-crossing count. Following Rabiner and Schafer (1978), the data is downsampled to 8192 kHz and broken into 40 ms blocks, which are analyzed by LP for the following characteristics: fundamental frequency, the first and second formant frequencies, and the bandwidth of each formant. RMS amplitude values are also calculated for each block of input. The values for each block of analysis are fed into a median smoothing filter (Rabiner and Schafer 1978: 158-161) to produce the low-level feature value for that analysis frame.

Performance of the LP code was a major concern of ours. So, in this case, we violated Knuth's maxim and prematurely optimized. The LP code is implemented in a slightly peculiar, non-object-oriented style. The goal was to minimize virtual and interface method lookup, and more importantly, to minimize object creation. Though such issues are often disregarded when writing Java, it should not be surprising that removing memory allocations in time-critical loops proved crucial to tuning this code. In the end, we were able to implement the signal analysis in pure Java with satisfactory performance.

3.2 MID-LEVEL ANALYSIS

The mid-level analysis parses the incoming low-level analysis data into gestures. Since users are asked to hold down a play button while they are making a sound, it was trivial to parse vocal input into gestures based on the button's press and release.[3] If the button is held down continuously for several seconds, the system will eventually create a gesture boundary to prevent any single gesture from becoming too long. And if there are periods of silence while the button remains down, the system will create additional gesture boundaries at those points.

Once a gesture is identified, a feature vector of statistical parameters is created to describe the entire gesture. The choice of features is based largely on studies of vocal signal analysis for emotion classification by Banse and Scherer (1996), Yacoub, Simske, Lin, and Burns (2003), and Cowie, Douglas-Cowie, Tsapatsoulis, Votsis, Kollias, Fellenz, and Taylor (2001). While we are not focused solely on emotion, we found this research a useful starting point. Studies of timbre, most of which extend Grey's (1977) multidimensional scaling studies, were also informative, but their focus on steady instrumental tones was less directly applicable to the variety of vocal gestures expected from Auracle users.

While many emotion classification studies try to separate linguistically determined features from emotionally determined features (Cowie et al. 2001), this is not necessary in Auracle. Our system responds to features of user input whether they are linguistically determined, emotionally determined, or consciously manipulated by users to control the instrument in specific ways.

Our mid-level feature vector includes 43 features: the mean, minimum, maximum, and standard deviation of f0, f1, f2, and RMS amplitude, as well as of their derivatives; the mean, minimum, maximum, and standard deviation of the durations of individual silent and nonsilent segments within the gesture; and the ratio of silent to nonsilent frames, voiced to unvoiced frames, and mean silent to mean nonsilent segment duration.

3.3 HIGH-LEVEL ANALYSIS[4]

It is theoretically possible to directly transmit each 43-element mid-level feature vector across the network and to map that vector onto synthesis control parameters, but we found it impractical in practice to directly address this amount of data.

Instead, we perform a high-level analysis which projects the 43-dimensional mid-level feature space onto 3 dimensions. In choosing those dimensions, we did not wish to merely select a subset of the mid-level features, nor did we wish to manually create projection functions: these approaches would have driven users to interact according to our own preconceptions, and in doing so would have contradicted the goals of the project.

We were attracted to the use of Principal Components Analysis (PCA) to generate this projection, because it preserves the greatest possible amount of variance in the original data set. In other words, the mid-level features which users themselves vary the most take on the greatest importance in the PCA projection. It facilitates a self-organizing, user-driven approach.

But PCA creates a static projection; for Auracle, we wanted a dynamic approach which could perform both short-term adaptation — by changing over the course of a single user session to focus on the mid-level features varied most by that user — and long-term adaptation, in which the classifier's initial state for each session slowly changes to concentrate on the mid-level features most varied by the entire Auracle user base.

An adaptive classifier does sacrifice a degree of transparency in its classifications: it is more difficult for users to relate their vocal gestures to sound output when the high-level feature classifications, and thus the mappings, are constantly changing. And it is impossible to interpret the meaning of high-level features during the design of mapping procedures, since their semantics change with adaptation. For us, though, transparency in this component of Auracle was less important than adaptability.

Our adaptive PCA implementation does not use classical PCA methods, in which the principal components of a set of feature vectors are the eigenvectors of the covariance matrix of the set with the greatest eigenvalues. This strategy is awkward to adapt to a continuously-expanding input set and computationally expensive to perform in real time in Java.

[Figure 2. APEX neural network.]

Instead, we implement the Adaptive Principal Component EXtraction (APEX) model (Diamantaras and Kung 1996 and Kung, Diamantaras, and Taur 1994), which improves upon earlier neural networks proposed by Oja (1982), Sanger (1989), Rubner and Tavan (1989), and others. APEX efficiently implements an adaptive version of PCA as a feed-forward Hebbian network (with modifications to maintain stability) and a lateral, asymmetrical anti-Hebbian network. The Hebbian portion of the network discovers the principal components, while the anti-Hebbian portion rotates those components. The learning rate of the algorithm is automatically varied in proportion to the magnitude of the outputs and a "forgetting" factor which controls the algorithm's memory of past inputs (Kung, Diamantaras, and Taur 1994).

Upon launching Auracle, a client's neural network is initialized with weights downloaded from the server.[5] The client-side neural network quickly adapts to the vocal gestures created by the local user, updating its internal weights accordingly. Then, when a user logs out of Auracle, the client's internal weights are transmitted back to the server, which merges them with its previous weight matrix to facilitate long-term adaptation.

Unlike many other neural networks, it is easy to monitor how APEX adapts; each feed-forward weight represents the importance of a particular mid-level feature in the computation of a particular high-level feature. This transparency was critical in developing, debugging, and evaluating the high-level analysis system within Auracle.

3.4 NETWORK

Each gesture's low-level analysis envelopes, along with the high-level feature values, are sent to a central server running TransJam (Burk 2000), a Java server for distributed music applications. The TransJam server provides a mechanism to create shared objects, acquire locks on those objects, and distribute notifications of changes to those objects. Each client sends its gesture data as a modification to a data object which it has locked, and the server then transmits the updated object information to all clients in the ensemble. In this manner, all client machines maintain all players' analysis data in sync.

By sending only control data, Auracle maintains low latency and high audio quality using a fraction of the bandwidth required for audio streaming. However, TransJam's XML text-based protocol is expensive when transmitting floating-point numbers, since each digit is sent as a separate ASCII character. So we compress those numbers using a fixed lookup table to dramatically improve bandwidth utilization.[6]

Java security restrictions and practical networking issues made direct peer-to-peer communication impossible. To mitigate the probability of a performance bottleneck, Auracle's architecture is designed to minimize the work done by the server. The server is merely a conduit for data and does no processing itself. Mapping and synthesis operations are duplicated by all clients, but we preferred this solution over adding load on the server. Our benchmarking shows that we can support 100 simultaneous users, each sending one gesture per second, with an average CPU load of only 35% on our Apple Xserve (G4 1.33 GHz, 512 MB RAM).

3.4.1 NETWORK LATENCY

The analysis data is transmitted to the server only once a complete gesture has been detected. This reduces network traffic and generally uses the network more efficiently. Data is only mapped onto synthesis control parameters when it arrives from the server, even when the data was created by the local client. This creates a short delay between the vocal input and synthesized response; we have found that this latency is not a disadvantage but rather facilitates a conversational style of interaction which works quite well.

3.4.2 EVENT DISTRIBUTION

Because it is impossible to predict exact network latency, and because user vocal gestures are not looped, it is difficult for Auracle users to plan vocal gestures to coincide with those of other players in an ensemble.

Rather than trying to synchronize gesture synthesis, we distribute the onset of gestures from different players to minimize their overlap. We add a small amount of additional delay before gesture onsets when it reduces overlap, and in dense textures, we also scale gestures so that their length is slightly shorter. As with our approach to network latency, our goal is to facilitate a conversational style of interaction, where players respond to past events rather than trying to synchronize with future events. As an additional benefit, users are more easily able to hear their contribution to the ensemble when gestures overlap less, increasing the system's transparency.

3.5 MAPPING AND SYNTHESIS

Each client receives the data from the server and passes it to a mapper, which turns the incoming data into synthesizer control parameters. The vocal gestures of each player control separate mapper and synthesizer instances.

[Figure 3. Synthesizer schematic.]

The synthesis algorithm, implemented entirely using the JSyn API (Burk 1998), is a hybrid of several techniques, designed to enable the mapping of player data onto a wide range of timbres. Two excitation sources —a pulse oscillator and a frequency-modulated sine oscillator — are mixed and sent through an extended comb filter with an averaging lowpass filter and probabilistic signal inverter included in the feedback loop. The result is sent through a bank of bandpass filters and mixed with the unfiltered sound to generate the final output.

Much of the low-level analysis data is mapped onto the synthesis algorithm in straightforward ways. The fundamental frequency envelope controls the frequency of the excitation sources and the length of the feedback delay line. The amplitude envelope controls the overall amplitude of the synthesizer. The first and second formant envelopes are used to set the center frequencies of the bandpass filters, and the Q on those filters are inversely proportional to the formant bandwidth envelopes.

Low-level analysis data is also used to tightly couple timbre changes with amplitude and frequency changes, in order to make these envelopes more salient in the synthesized sound and to make the user’s contribution to those sounds more transparent. The amplitude envelope is used to control the depth of the frequency modulation, and the fundamental frequency envelope controls the ratio of frequency modulation.

High-level feature data, on the other hand, is used to control timbral aspects of the synthesis which evolve from one gesture to the next but do not change within a single gesture: the probability of inverting the feedback signal and the ratio of pulse to sine generators in the excitation source. And sometimes, both low-level and high-level data is combined to influence a single aspect of the synthesis algorithm: one high-level feature defines the range within which the formant bandwidths modify the filter Q values during each gesture.

We did not want sound output to stop completely when users were not making vocal gestures. So when a player's synthesizer is finished playing a gesture, it continues sounding a quiet "after ring" until the next user gesture is received. The relationship of the vocal gesture to this after ring is less transparent than with the gesture itself; it is designed simply to be a quiet sound which constantly but subtly changes. It is based on the formant envelopes of the previous gesture, slowed down dramatically and played out of phase with each other.

3.6 GRAPHICAL USER INTERFACE

[Figure 4. Auracle Graphical User Interface.]

The focus of Auracle is on aural interaction, so the software's graphical user interface is deliberately sparse. The main display area shows information about all users in the active ensemble of players: their usernames, their approximate locations on a world map (computed with an IP-to-location service), and a running view of the gestures they make (displayed as a series of colored squiggles corresponding to their amplitude, fundamental frequency, and formant envelopes).

Users push and hold a large play button when they want to make a vocal gesture. Additional controls allow them to move to another ensemble, create a new ensemble, and monitor and adjust audio levels. A text chat among players within the ensemble is available in a separate popup window.

4 DISTRIBUTED DEVELOPMENT PROCESSES

Not only is Auracle itself is a collaborative, networked instrument, but it was developed through a collaborative, networked process. The six-member project team had members based in Germany, Italy, California, and Arizona. During the year-long development process, the team met for three intensive week-long meetings in Germany to discuss key aesthetic and architectural issues. But the deployment of standard collaboration and communication tools was essential in coordinating the efforts of team members throughout the year and in making the project a reality.

4.1 PROJECT-SPECIFIC DEVELOPMENT TOOLS[7]

4.1.1 DYNAMIC SYSTEM CONFIGURATION

We designed Auracle as a component-based architecture because we wanted to experiment with a variety of approaches, particularly with regards to mapping and synthesis techniques. The final release of Auracle only uses a small fraction of the hundreds of components we created.

Auracle's architecture uses Java interfaces, reflection, and the observer pattern, combined with an avoidance of direct cross references, so that components can be mixed and matched to form a complete system. During startup, the application reads a text file specifying the particular components to be used and instantiates the corresponding configuration.

Reconfiguration of Auracle does not require the source code to be recompiled, but it does require the configuration file to be edited and the program to be restarted. Rapid comparisons between configurations are not possible. And small tweaks to synthesizer parameters require changes to the source code; they cannot be specified in the configuration file. As the number of experimental components grew, tracking and comparing components and configurations became increasingly difficult, and manually distributing configurations to colleagues became tedious.

[Figure 5. Auracle TestBed graphical user interface.]

To address these limitations, we created the Auracle Testbed, a separate application used only in the development process and not included in the public release. Popup menus in the Testbed's GUI select analyzer, mapper, synthesizer, and effects unit components, and sliders adjust internal synthesizer parameters for fine-tuning control.

The Testbed saves configurations of analyzers, mappers, synthesizers, and effects units as patches. Developers annotate patches through name and description fields to add comments or help explain them to other team members. The patches are saved as text files and also displayed as buttons in the graphical user interface. A single button press switches to a different system configuration, enabling rapid comparisons between patches. The change in Auracle's configuration is immediate; no text files need to be edited and the application does not need to be restarted.

From within the Testbed, developers can also easily upload patches to the group development server to share them with other team members, who can use them in a group “jam session” or download them to their local machine.

4.1.2 INTEGRATION WITH EXISTING TOOLS

The Auracle Testbed can also send analysis data to any application which supports the Open Sound Control (OSC) protocol (Wright and Freed 1997). We used this feature to send Auracle data in real time to SuperCollider, Max/MSP, and Wire. By combining Auracle with external sound development tools, we were able to quickly prototype new ideas using existing synthesis libraries and user-friendly environments which permitted runtime modifications to synthesis algorithms.

We exported synthesis patches developed in Wire as Java source code and directly integrated them into Auracle’s Java source tree. For synthesis algorithms designed in the other applications, we manually ported the most successful algorithms to Java, which was straightforward.

4.2 REMOTE COLLABORATION TOOLS

We integrated a variety of online collaboration tools into our workflow to ensure that our vision of the project remained in sync, our work schedules were coordinated, and our priorities were clear. These included both structured collaboration tools — a bug tracking database and group task and calendar software — and unstructured environments — a

Wiki for collaborative development of project documents and a mailing list (with searchable archives) for free-form discussion.

Equally important, Auracle itself became a platform for our own collaboration on the project. We quickly developed prototypes for all the components in the architecture, along with text-based chat functionality, and began holding twice-weekly “jam sessions” on our development builds. These jams, which were usually followed by Internet-based audio conference calls, were critical opportunities to track our progress and identify technical and aesthetic issues. They also helped us to regularly experience Auracle as users rather than as developers.

4.3 EXTREME PROGRAMMING PRACTICES

Networked software development necessitated the use of good programming and development habits to keep our code clear, integrated, and synchronized. We followed many of the development practices encouraged by the Extreme Programming (XP) paradigm (Beck 1999), including nightly automated builds and unit testing on our development server, and frequent developer collaboration and task rollover from one developer to another.

4.3.1 AUTOMATING USER INPUT

Since Auracle is a voice-controlled instrument, we needed to constantly create vocal sounds in both manual and automated testing situations. Our TestBed application enables us to quickly select and loop through audio files which replace microphone input into Auracle, and we maintained a large vocal gesture sample database to use in this regard. A second, smaller collection of sound files documented gestures which caused problems such as inaccurate analyses, overloaded synthesis filters, or even crashes. We used these files to consistently reproduce problems as we were trying to fix them. Sound file playback was also incorporated into our automated unit testing architecture.

Auracle is designed for use by an ensemble of participants, so it was important to test it in group situations throughout the development process. Mapping and synthesis components sounded dramatically different when used individually than when used in a group "jam session." Many bugs only occurred in group situations. And we also needed to test the server under heavy loads to benchmark performance and determine capacity.

To address these needs, we developed a Headless Client to simulate the activity of a single user. In order to reduce CPU usage, the Headless Client pre-analyzes audio files and stores data in a form ready to transmit to the server. It references this preprocessed data when "jamming" on Auracle. And it does not perform any mapping or synthesis on data received back from the server.

A command-line application launches several Headless Clients simultaneously to simulate one or more ensembles of participants. A developer can simulate dozens of users from a single machine and then launch a single instance of the complete applet to "jam" with them interactively.

4.3.2 DOCUMENTATION

We used Javadoc functionality to create self-documenting code; Javadoc web pages were updated nightly as part of our nightly build process. We complemented these Javadocs with higher-level component architecture descriptions, which were posted and updated manually on our Wiki.

4.3.3 DEBUGGING AND TRACKING MECHANISMS

Once we released a beta version of Auracle to the public, we wanted to monitor user activity to identify the problems users encountered. A combination of several different logging mechanisms track this information.

Web server logs provide basic information about site visitors, and the TransJam server tracks some rudimentary information about user sessions. But this was little help when trying to find the source of reported problems or trying to track unreported issues.

So the Auracle applet complements this data by uploading more detailed information to a server-side database, tracking each client's operating system, web browser, Java implementation, and any client-side error messages and Java stack traces generated during the session. The database is searchable via a web interface, and daily e-mail summaries are sent to our mailing list.

This logging data helps us more easily track and fix bugs. When users send us problem reports, we can quickly locate their session in the database and find information about their system configuration and any errors which Auracle logged; they do not need to figure out these details themselves. We can also look directly in the database to find errors which were never reported by users at all. Often, a stack trace in the log points us to a specific line of source code and an easy solution.

5. DISCUSSION

Auracle was officially launched to the public in October 2004 — on the Internet, at Donnaueschinger Musiktage in Germany, and during a live radio event on SWR. Since then, we have received feedback from numerous Internet-based Auracle users, and we have watched people interact with Auracle and discussed their experiences with them at several events where Auracle kiosks have been installed.

We have been thrilled to see how Auracle engages people ranging from nonmusicians to trained singers, of many different ages and cultural backgrounds. Many users are drawn into extended interactions with the system, and it is always surprising to hear the variety of vocal sounds they create and the variety of sounds the system creates in response.

We are also pleased with the long-term adaptation of the high-level classification. We often return to the system ourselves after a week or two and feel noticeable changes in its timbral response to our voices. We would still like to find additional ways to make the system adapt to user activities over time and learn from what they do.

5.1 VIABLE USER BASE

The biggest challenge we face with Auracle is to expand its user base. During the four months beginning October 15, 2004, there were 1097 user sessions on Auracle, with 590 usernames connecting from 520 distinct hosts. Auracle is most interesting when users are online at the same time and can "jam" together, but most of these users were alone when they used Auracle.[8] We want to attract a large enough user base so that users consistently find other players online.

We have experimented with a variety of strategies, including scheduling specific online events and enabling Auracle users to easily schedule online meetings with friends, but these techniques have met with limited success. Our most successful Auracle events, ironically, have been physical, not virtual, events: several computers are set up as kiosks on which people can try Auracle. We are continuing to present Auracle in this format, and we are also exploring the possibility of permanent kiosks in museums and other public spaces.

While it is difficult to draw users to visit the site, it is even more challenging to get them set up and logged in once they arrive. We designed Auracle with easy setup in mind, and we tested extensively for compatibility on a wide variety of platforms and configurations.

But the statistics are disappointing: during the four-month period beginning October 15, 2004, 6,052 distinct hosts visited the Auracle web site, yet only 520 of those hosts actually launched and logged in to Auracle. And while users who do log in are engaged for long periods of time — an average of 18 minutes — 55% of users never actually input a sound into Auracle at all. While some of those users are likely perplexed by the user interface or are too shy to contribute, our informal polling indicates that the majority of them simply lack computer microphones.

There is little we can do about this problem; it is unreasonable to expect users to buy an external microphone or headset just to use Auracle, But we are encouraged by the growing popularity of online audio chat and telephony applications, and we hope that computer microphones will soon be ubiquitous even on desktop machines.

5.2 OPENING AURACLE TO THE COMPUTER MUSIC COMMUNITY

Auracle was designed for a lay public without formal musical or technical training, and those users have been the focus of our efforts to date. Now, we want to make the project more accessible to members of the computer music community. We are preparing much of the Java source code for release under an open-source license, so that others may leverage our development work in their own projects. We are also developing a Software Developer's Kit which will enable developers to create their own mappers and synthesizers to contribute for use within Auracle. By opening Auracle development to new contributors, we hope that the project will evolve in new ways and new directions we could not have envisioned ourselves.

BIBLIOGRAPHICAL REFERENCES

Antares audio technologies. Antares kantos. 2004. http://www.antarestech.com/products/kantos.html.

Banse, R., and K. Scherer. 1996. Acoustic Profiles in Vocal Emotion Expression. *Journal of Personality and Social Psychology*, 70 (3): 614-636.

Barbosa, A. 2003. Displaced Soundscapes: A Survey of Networked Systems for Music and Sonic Art Creation. *Leonardo Music Journal* 13: 53-59.

Barbosa, A. and M. Kaltenbrunner. 2002. Public Sound Objects: A Shared Musical Space on the Web. *Proceedings of International Conference on Web Delivering of Music 2002.* Darmstadt, Germany, IEEE Computer Society Press: 9-15.

Beck, K. 1999. *Extreme programming Explained: Embrace Change.* Reading, MA, Addison-Wesley.

Böhlen, M., and J. Rinker. 2004. When Code is Context: Experiments with a Whistling Machine. *Proceedings of the 12th ACM International Conference on Multimedia*. New York: ACM, 983-984.

Brown, C. 2003. Eternal Network Music. http://crossfade.walkerart.org/brownbischoff2/.

Bryan-Kinns, N. and P. Healey. 2004. DaisyPhone: Support for Remote Music Collaboration. *Proceedings on the 2004 Conference on New Interfaces for Musical Expression*. Hamamatsu, Japan, ACM: 29-30.

Burk, P. 1998. JSyn – A Real-time Synthesis API for Java. *Proceedings of the 1998 International Music Conference*. Ann Arbor, MI: ICMA, 252-255.

Burk, P. 1999. WebDrum. http://www.transjam.com/webdrum/.

Burk, P. 2000. Jammin' on the web – a new client/server architecture for multi-user performance. *Proceedings of the 2000 International Music Conference*. Berlin, Germany: ICMA, 117-120.

Cowie, R., E. Douglas-Cowie, N. Tsapatsoulis, G. Votsis, S. Kollias, W. Fellenz, and J. Taylor. 2001. Emotion Recognition in Human-Computer Interaction. *IEEE Signal Processing Magazine*, January 2001, 32-80.

Diamantaras, K. and S. Y. Kung. 1996. *Principal Component Neural Networks*. New York: John Wiley and Sons, Inc.

Duckworth, W. 2000. The Cathedral Project. http://cathedral.monroestreet.com/.

Freeman, J., C. Ramakrishnan, K. Varnik, M. Neuhaus, P. Burk, and D. Birchfield. 2004. Adaptive High-level Classification of Vocal Gestures Within a Networked Sound Environment. *Proceedings of the 2004 International Computer Music Conference*. Miami, FL, ICMA: 668-671.

Grey, J. 1977. Multidimensional perceptual scaling of musical timbres. *Journal of the Acoustical Society of America*, 61 (5): 1270-1277.

Joyce, D. 2005. Get Your Own Show. http://www.negativland.com/nmol/ote/text/getoshow.html.

Kung, S., K. Diamantaras, and J. Taur. 1994. Adaptive Principal Component EXtraction (APEX) and Applications. *IEEE Transactions on Signal Processing*, 42 (5), 1202-1217.

Machover, T. 1996. The Brain Opera. http://brainop.media.mit.edu.

Neuhaus, M. 1994. The Broadcast Works and Audium. http://www.kunstradio.at/ZEITGLEICH/CATALOG/ENGLISH/neuhaus2-e.html.

Oja, E. 1982 A Simplified Neuron Model as a Principal Component Analyzer. *Journal of Mathematical Biology,* 15: 267-273.

Oliver, W. 1997. *The Singing Tree, A Novel Interactive Musical Interface*. M.S. thesis, EECS Department, Massachusetts Institute of Technology.

Rabiner, L. and R. Schafer. 1978. *Digital Processing of Speech Signals*. Englewood Cliffs, NJ, Prentice-Hall.

Radio Show Calling Tips. 2005. http://www.pressthebutton.com/calling.htm.

Ramakrishnan, C., J. Freeman, K. Varnik, D. Birchfield, P. Burk, and M. Neuhaus. 2004. The Architecture of Auracle: A Real-Time, Distributed, Collaborative Instrument. *Proceedings of the 2004 Conference on New Interfaces for Musical Expression*. Hamamatsu, ACM: 100-103.

Rubner, J., and P. Tavan. 1989. A Self-Organizing Network for Principal-Components Analysis. *Europhyics. Letters*, 10(7): 693-698.

Sanger, T. 1989. An Optimality Principle for Unsupervised Learning. In D. Touretzky (ed.) *Advances in Neural Information Processing Systems*. San Mateo, CA: Morgan Kaufman.

Tanaka, A. 2000. MP3Q. http://fals.ch/Dx/atau/mp3q/.

The User. 2000. Silophone. http://www.silophone.net.

Varnik, K., J. Freeman, C. Ramakrishnan, P. Burk, D. Birchfield, and M. Neuhaus. 2004. Tools Used Whlie Developing Auracle: A Voice-Controlled, Networked

Instrument. *Proceedings of the 12th ACM International Conference on Multimedia*. New York: ACM, 528-531.

Wright, M. and A. Freed. 1997. Open sound control: A new protocol for communicating with sound synthesizers. *Proceedings of the International Computer Music Conference*. Thessaloniki, Hellas, ICMA, 101-104.

Yacoub, S., S. Simske, X. Lin, and J. Burns. 2003. Recognition of Emotions in Interactive Voice Response Systems. http://www.hpl.hp.com/techreports/2003/HPL-2003-136.html.

CAPTIONS

Figure 1. Auracle system architecture.

Figure 2. The APEX neural network as used within Auracle. X nodes represent mid-level features (input) and Y nodes represent high-level features (output). W weights are feed-forward, C weights are lateral.

Figure 3. Synthesizer schematic.

Figure 4. Auracle graphical user interface.

Figure 5. Auracle TestBed graphical user interface.

NOTES

---

[*] The Auracle project is a production of Akademie Schloss Solitude with financial support from the Landesstiftung Baden-Würtemburg. We express our gratitude at their generous support. Auracle is available at http://auracle.org.

[1] For an extended discussion, see Ramakrishnan, Freeman, Varnik, Birchfield, Burk, and Neuhaus 2004.

[2] This analysis is predicated on the assumption that the incoming sound is vocal. We are not guaranteed that the user is making vocal sounds, but we treat all input as if it were vocal.

[3] Our primary motivation for this interface design was to reduce feedback in the system, in which audio output was re-input through the microphone as a new vocal gesture.

[4] For an extended discussion, see Freeman, Ramakrishnan, Varnik, Neuhaus, Burk, and Birchfield 2004.

[5] The server-side weights are initialized through training on a database of 230 recorded vocal gestures created by ten participants.

[6] Since we know all data is numerical, it is trivial to devise such a lookup table.

[7] For an extended discussion, see Varnik, Freeman, Ramakrishnan, Burk, Birchfield, Neuhaus 2004.

[8] We have created perpetual, virtual ensembles on Auracle where users can interact with robots if they wish.