We present libCollider [3], a client library for SuperCollider’s sclang sound synthesis engine that provides C++ application developers with direct access to sclang’s sophisticated capabilities for real-time audio synthesis and rendering through an API with abstracted bi-directional network support. Its development is driven by the demand for an audio component to the CalVR visualization framework under development at the Immersive Visualization Laboratory at the Qualcomm Institute, San Diego. We describe common problems with audio integration into C++ real-time graphics applications, and the specific demands we are trying to meet with libCollider’s multi-level API.

1. WHAT’S IN THE WAY OF AUDIOVISUAL INTEGRATION IN REAL-TIME RENDERING?

Sound is an important sensory modality of human experience, but even in 2013, it is often still addressed very insufficiently by applications for data and information display. A coherent integration of audio components into visualization environments has much promise—some of which can be discovered in Michel Chion’s book Audio-Vison [8]. However, attempts to tap into these potentials of inter-modal perception have to overcome a variety of obstacles. First of all, in the interdisciplinary discourse that surrounds the development of visualization systems, audio components tend to occupy the role of a peripheral add-on, as something that comes into perspective after the work on the visualization has already been successfully completed. On the technical level, the resources left to meet the demands of qualitative sound projection are often scarce—the need for good loudspeakers and their placement, of low ambient noise around the display, considerations of room acoustics and screen reflections, etc.

But next to the lack of opportunity to experience high-quality sound projections first person and of awareness for the potential roles of audio in audio-visual display, there are also important structural differences between image and sound that tend to be overlooked. Even though auditory and visual media components have the potential to generate unified audio-visual impressions, the respective senses are characterized by a different responsiveness to temporal development, which results in divergent requirements for form and structure of their display and their digital rendering. Music and sound diverge in their temporal morphology from the dynamic behaviors of visual images and animation. This results in divergent demands for temporal behavior and accuracy of perception-oriented processes. Even though the temporal morphology of respective real-time processing in general. A separation of the real-time computation for both modalities into independent parallel processes is unavoidable, and as a result, audio-visual software projects tend to be assembled on separate development platforms, which in turn reflects in a segregation of the respective development teams, severely complicating the creation of audio-visual potentials.

We can also see this problem reflected in the emergence of applicable standards.

1.1. Standards and their adoption

In the visual domain, open programming interfaces such as OpenGl and geometric scene graphs [4] have been adopted across a wide variety of platforms. Similar attempts in the audio domain, such as MPEG-4 and OpenNl, for example, have not developed a similar universal appeal and have not transcended specific application niches. Even less is heard about unified audio-visual platform standards.

In the late 1990s it seemed as if sound and image were coming together in programming environments such as Max/MaxMSP/Pitter and PD/Gem, but these environments have mostly been used by musicians to do visuals, while visualization researchers and visual artists continued to pursue sound-independent infrastructures like Processing, openFrameworks et cetera, that rely on the definition of a custom network protocol to connect to an independent real-time audio programming environment. While the potential to share data between independent programming environments through a network protocol such as OSC [16] opened up many possibilities, the resulting structural complexity of independent components is also a source of much friction and frustration.

Next to establishing the network protocol as such, the two communicating environments need to be kept mutually in sync, and often specialist programmers—often one for the visual and one for the audio domain—are required to operate in close communication. This lack of a unified coding platform for visual and auditory rendering stands in the way of the successful design of detailed inter-modal experiences.

2. SOME EXISTING STRATEGIES

In Table 1, we list a small array of real-time audio rendering infrastructures that are used in application development today.

2.1. Integrated vs. standalone API

Auditory and visual components of interactive applications need to operate in temporal coherence, but in symmetry to their distinct physiological and cognitive processing, rendering and interactive adaptation are best addressed by distinct processing architectures. For example, vision-oriented processed usually proceeds in frames that are updated every 40 or so milliseconds, while audio processing attempts to produce a continuous stream of much finer temporal resolution, without the inherent need of frame-oriented temporal subdivisions. While most graphically processing is spatial—transforming 2D images or rendering perspectives on 3D geometry—audio processing usually operates on a set array of correlated time-series and their continuous temporal evolution. Adding real-time audio to a visual application therefore implies the addition of a distinct temporal processing architecture, and the solutions can be roughly divided into two categories:

1. running a connected stand-alone programming environment for real-time audio in parallel to the core application,
2. including a suitable real-time audio library into the application’s native code.

2.2. Standalone-API solutions

As an alternative to the creation of a detached SoundServer process, it is also possible to include audio libraries into the core application directly, consolidating the sound design with the application development while dispensing with the requirement for system-level coordination of independent client and server components.

Most of the integrated audio libraries available today—OpenAL, FMOD Ex, irrKlang to name a few—provide functionality for a specific application domain, for example video game applications. Shortcomings are often a lack of flexibility in the channel configuration, routing, processing graphs. Examples are Pure Data, Max/MSP, Audio Mulch, CSound, and SuperCollider.

Ideally a server implemented in one of these languages runs in a parallel processing thread on the host machine or on an independent remote machine controlled by a bi-directional network protocol of network packets, either direct UDP or Open Sound Control [16]. This separation of audio client and server also permits the use of specialized machine hardware and software that helps to manage the overall CPU load.

In order to be used with a separate application framework like our target application framework CalVR [13], communication protocols are defined that control the audio processing graph in the remote sound process. Previous efforts in this direction include Gerhard Eckel’s SoundServer [9], and we have undertaken various efforts at our own institute to provide visualization and VR applications with MaxMSP and PD-based sound-server components that communicate to the visual application using specifically created OSC-based network protocols [14, 15].
AUDIOVISUAL INTEGRATION WITH CALVR AND LIBCOLLIDER

Eric Hamdan, Joachim Gossmann
Qualcomm Institute
La Jolla, CA, U.S.A.
hamdan@eng.ucsd.edu
gossmann@ucsd.edu

ABSTRACT

We present libCollider [3], a client library for SuperCollider’s sclang sound synthesis engine that provides C++ application developers with direct access to sclang’s sophisticated capabilities for real-time audio synthesis and rendering through an API with abstracted bi-directional network support. Its development is driven by the demand for an audio component to the CalVR visualization framework under development at the Immersive Visualization Laboratory at the Qualcomm Institute, San Diego. We describe common problems with audio integration into C++ runtime graphics applications, and the specific demands we are trying to meet with libCollider’s multi-level API.

1. WHAT’S IN THE WAY OF AUDIOVISUAL INTEGRATION IN REAL-TIME RENDERING?

Sound is an important sensory modality of human experience. Even though in 2013, it is often still addressed very insufficiently by applications for data and information display.

A coherent integration of audio components into visualization environments has much promise—some of which can be discovered in Michel Chion’s book Audio-Vision [8]. However, attempts to tap into these potentials of intermodal perception have to overcome a variety of obstacles.

First of all, in the interdisciplinary discourse that surrounds the development of visualization systems, audio components tend to occupy the role of a peripheral add-on, as something that comes into perspective after the work on the visualization has already been successfully completed.

On the technical level, the resources left to meet the demands of qualitative sound projection are often scarce—the need for good loudspeakers and their placement, of low ambient noise around the display, considerations of room acoustics and screen reflections, et cetera.

Additionally, the expressive potentials of multi-channel sound systems as they have been explored in the area of computer music and more recently multichannel movie soundtracks often remain unconsidered in the planning and conceptualization of the display as do the manifold potentials to display data through non-speech sound [10].

But next to the lack of opportunity to experience high quality sound projections first person and of awareness for the potential roles of audio in audio-visual display, there are also important structural differences between image and sound that tend to be overlooked.

Even though auditory and visual media components have the potential to generate unified audio-visual impressions, the respective senses are characterized by a different responsiveness to temporal development, which results in divergent requirements for form and structure of their display and their digital rendering.

Music and sound diverge in their temporal morphology from the dynamic behaviors of visual images and animation. This results in divergent demands for temporal behavior and accuracy of perception-oriented process scheduling, and the temporal morphology of respective real-time processing in general. A separation of the real-time computation for both modalities into independent parallel processes is unavoidable, and as a result, audio-visual software projects tend to be assembled on separate development platforms, which in turn is reflected in a segregation of the respective development teams, severely complicating the creation of audio-visual potentials.

We can also see this problem reflected in the emergence of applicable standards.

1.1. Standards and their adoption

In the visual domain, open programming interfaces such as OpenGL and geometric scene graphs [4] have been adopted across a wide variety of platforms. Similar attempts in the audio domain, such as MPEG-4 and OpenAL for example, have not developed a similar universal appeal and have not transcended specific application niches. Even less is heard about unified audio-visual platform standards.

In the late 1990s it seemed as if sound and image were coming together in programming environments such as Max/MSP/Jitter and PD/Gem, but these environments have mostly been used by musicians to do visuals, while visualization researchers and visual artists continued to pursue sound-independent infrastructures like Processing, openFrameworks et cetera, that rely on the definition of a custom network protocol to connect to an independent real-time audio programming environment.

While the potential to share data between independent programming environments through a network protocol such as OSC [16] opened up many possibilities, the resulting structural complexity of indepent components is also a source of much friction and frustration.

Next to establishing the network protocol as such, the two communicating environments need to be kept mutually in sync, and often specialist programmers—one for the visual and one for the audio domain—are required to operate in close communication.

This lack of a unified coding platform for visual and auditory rendering stands in the way of the successful design of detailed inter-modal experiences.

2. SOME EXISTING STRATEGIES

In Table 1, we list a small array of real-time audio rendering infrastructures that are used in application development today.

2.1. Integrated vs. standalone API

Auditory and visual components of interactive applications need to operate in temporal coherence, but in symmetry to their distinct physiological and cognitive processing, rendering and interactive adaptability are best addressed by distinct processing architectures. For example, vision-oriented processing usually proceeds in frames that are updated every 40 or so milliseconds, while audio processing attempts to produce a continuous stream of much finer temporal resolution, without the inherent need of frame-oriented temporal subdivisions. While most graphics processing is spatial—transforming 2D images or rendering perspectives on 3D geometry—audio processing usually operates on a set array of correlated time-series and their continuous temporal evolution.

Adding real-time audio to a visual application therefore implies the addition of a distinct temporal processing architecture, and the solutions can be roughly divided into two categories:

1. running a connected stand-alone programming environment for real-time audio in parallel to the core application,
2. including a suitable real-time audio library into the application’s native code.

2.2. Standalone-API solutions

In a frequently encountered scenario in category 1), a library of audio functionalities is implemented in the audio programming language as a Sound-Server that provides services such as audio playback and synthesis.

Often we find sound servers implemented in standalone domains of specific programming environments that provide libraries of primitives and abstractions for audio synthesis and data processing that can be combined into audio processing graphs. Examples are Pure Data, Max/MSP, Audio Mulch, CSound, and SuperCollider.

Ideally a server implemented in one of these languages runs in a parallel processing thread on the host machine or on an independent remote machine controlled by a bi-directional protocol of network packets, either direct UDP or Open Sound Control [16]. This separation of audio client and server also permits the use of specialized machine hardware and software that helps to manage the overall CPU load.

In order to be used with a separate application framework like our target application framework CalVR [13], communication protocols are defined that control the audio processing graph in the remote sound process. Previsus efforts in this direction include Gerhard Eckel’s SoundServer [9], and we have undertaken various efforts at our own institute to provide visualization and VR applications with Max/MSP and PD-based sound-server components that communicate to the visual application using specifically created OSC-based network protocols [14, 15].

Figure 1: Connecting the application core to a standalone audio-API with a custom communication protocol.

The use of standalone audio synthesis solutions requires the maintenance of the coherence between the client and the server component, forcing the developer of the client application to deal with an environment they may not be specialized in (or prefer to stay away from). In certain situations the management of domain-specific code such as SCLang or abstractions in PD and Max/MSP might be unacceptable. A suitable library and API in the native language of the client code may be preferred, which leads us to category 2) - integrated audio solutions.

2.3. Integrated solution

As an alternative to the creation of a detached SoundServer process, it is also possible to include audio libraries into the core application directly, consolidating the sound design with the application development while dispensing with the requirement for system-level coordination of independent client and server components.

Most of the integrated audio libraries available today—OpenAL, FMOD Ex, irrKlang to name a few—provide functionality for a specific application domain, for example video game applications. Shortcomings are often a lack of flexibility in i/o channel configuration, routing,
multichannel rendering, but most of all strong limitations on dynamic real-time audio synthesis.

3.2. Portable, light, efficient and easy to install

The demands for audio support arise on different platforms, computing infrastructures and application models. This places specific demands on cross-platform portability, computational efficiency and independence from platform-specific libraries. Server and client should be able to run on at least Mac OS, Linux, Windows, potentially Android and iOS, and should provide support for virtual reality displays, immersive environments, desktop apps, mobile apps, large scale display walls, unexpected art installations as well as computer-based audiovisual instruments, performances, et cetera.

3.3. Intuitive, multi-level API

We want our programming library to accommodate a range of different approaches to audio programming. On the one hand, standardized functionality such as the controls for localized sound playback needs to be made accessible in a convenient way that does not require a full knowledge of the API details. On the other hand, we want to equip our API with the potential to control and modify the audio processing and synthesis in way comparable to the standalone APIs we have previously used. In order to enable powerful interactive sound design, the server should support responsive and adaptive real-time synthesis with controllable filters, modulators and other real-time processing units such as those found in SuperCollider, Max/MSP, Pure Data, etc.

Natural this leads us to the requirement of a multi-level API that provides full low-level access as well as encapsulated functionality that can be used without specialized knowledge. The API should be completely sufficient to control audio rendering and make additional specialized programming environments unnecessary.

3.4. Flexible and interchangeable support for different multichannel techniques

Industry standard C++ libraries for sound design rarely offer flexible and extensible multichannel support used in experimental speaker layouts. Since we are exploring different spatial audio rendering techniques such as Wavefield Synthesis, VBAP, Ambisonics, Beamforming, et cetera, we need our Audio API to integrate the respective rendering easily. We need support for multichannel spatial audio rendering algorithms, flexible multichannel routing, and easily managed multichannel objects.

3.5. Open source audio backend

The server should be, or be built on, a well maintained open source software for accomplishing the tasks listed above to ensure that projects created now will see continued improvement over time as the server improves from community development, since our current personnel will not be able to maintain a proprietary custom solution on its own.

4. WHY SUPERCOLLIDER?

SuperCollider is a computer music programming language developed by James McCartney. In its third implementation, also known as sc3, McCartney split the system architecture into two independent components—a language process that permits a temporal structuring and organization of the processing, and an independently running synthesis engine. The two components are coupled via an internal network message protocol implemented in Open Sound Control [16]. This makes `scsynth`, the component

5. HOW IS LIBCOLLIDER DIFFERENT FROM LIBPD?

libpd is a software library that allows one to run an instance of Pure Data [5] as an audio processing callback and to allow MIDI and control message communication between application code and Pure Data code [7, 11]. Programmers can create a PD patch, load it into an instance of Pure Data running in a parallel thread to the application proper, and address it as a custom audio rendering engine by communicating to it through a set of language bindings provided by libpd. While this approach has some aspects in common with the way libCollider uses `scsynth`, there are important differences. On the hand libCollider is currently only accessible from C++, while libpd implements bindings for an array of additional languages such as Java, Objective-C, Python, et cetera. On the other hand, libCollider attempts to make the functionality provided by SuperCollider’s `scsynth` completely accessible from an external API, with no need for an additional graphical or text-based patching language such as the graphical patching of Pure Data or SCLang. Instead of delegating the low-level audio programming to an external language, libCollider organizes its functionality into different layers, from low-level functionality that wraps the commands available to control `scsynth`, to mid-level classes that allow a use of C++ in a similar way as SCLang, to a high-level API layer that provides the programmer with a simplified interface to often-used and basic audio functionality comparable to what a library like OpenAL provides. The multi-level layout of the API is further explained in section.

The architecture libCollider proposes has the advantage that the audio rendering process can reside on a different machine connected by network messages, thereby providing the advantages of a standalone Sound-Server
multichannel rendering, but most of all strong limitations on dynamic real-time audio synthesis.

3.2. Portable, light, efficient and easy to install
The demands for audio support arise on different platforms, computing infrastructures and application models. This places specific demands on cross-platform portability, computational efficiency and independence from platform-specific libraries: Server and client should be able to run on at least Mac OS, Linux, Windows, potentially Android and iOS, and should provide support for virtual reality displays, immersive environments, desktop apps, mobile apps, large scale display walls, unexpected art installations as well as computer-based audiovisual instruments, performances, etc. et cetera.

3.3. Intuitive, multi-level API
We want our programming library to accommodate a range of different approaches to audio programming. On the one hand, standardized functionality such as the controls for localized sound playback needs to be made accessible in a convenient way that does not require a full knowledge of the API details. On the other hand, we want to equip our API with the potential to control and modify the audio processing and synthesis in way comparable to the stand-alone APIs we have previously used. In order to enable powerful interactive sound design, the server should support responsive and adaptive real-time synthesis with controllable filters, modulators and other real-time processing units such as those found in SuperCollider, Max/MSP, Pure Data, etc.

Naturally this leads us to the requirement of a multi-level API that provides full low-level access as well as encapsulated functionality that can be used without specialized knowledge. The API should be completely sufficient to control audio rendering and make additional specialized programming environments unnecessary.

3.4. Flexible and interchangeable support for different multichannel techniques
Industry standard C++ libraries for sound design rarely offer flexible and extensible multichannel support used in experimental speaker layouts. Since we are exploring different spatial audio rendering techniques such as Wavefield Synthesis, VBAP, Ambisonics, Beamforming, etc., we need our Audio API to integrate the respective rendering easily. We need support for multichannel spatial audio rendering algorithms, flexible multichannel routing, and easily managed multichannel objects.

3.5. Open source audio backend
The server should be, or be built on, a well maintained open source framework for accomplishing the tasks listed above to ensure that projects created now will see continued improvement over time as the server improves from community development, since our current personnel will not be able to maintain a proprietary custom solution on its own.

4. WHY SUPERCOLLIDER?
SuperCollider is a computer music programming language developed by James McCartney. In its third implementation, also known as SCLang, McCartney split the system architecture into two independent components—a language process that permits a temporal structuring and organization of the processing, and an independently running synthesis engine. The two components are coupled via an internal network message protocol implemented in Open Sound Control [16]. This makes scsynth, the component

5. HOW IS LIBCOLLIDER DIFFERENT FROM LIBPD?
Libpd is a software library that allows one to run an instance of Pure Data [5] as an audio processing callback and to allow MIDI and control message communication between application code and Pure Data code [7, 11]. Programmers can create a PD patch, load it into an instance of Pure Data running in a parallel thread to the application proper, and address it as a custom audio rendering engine by communicating to it through a set of language bindings provided by libpd. While this approach has some aspects in common with the way libCollider uses scsynth, there are important differences. On the one hand libCollider is currently only accessible from C++, while libpd implements bindings for an array of additional languages such as Java, Objective-C, Python, etc. et cetera. On the other hand, libCollider attempts to make the functionality provided by SuperCollider’s scsynth completely accessible from an external API, with no need for an additional graphical or text-based patching language such as the graphical patching of Pure Data or SCLang. Instead of delegating the low-level audio programming to an external language, libCollider organizes its functionality into different layers, from low-level functionality that wraps the commands available to control scsynth, to mid-level classes that allow a use of C++ in a similar way as SCLang, to a high-level API layer that provides the programmer with a simplified interface to often-used and based audio functionality comparable to what a library like OpenAL provides. The multi-level layout of the API is further explained in section .
solution, while at the same time providing a close coupling between the sound rendering process and the application programming interface.

libpd achieves a separation of the application thread from the sound rendering thread, but does in itself neither permit an on-the-fly reorganization of audio rendering, nor—without the addition of a custom network layer—to displace the sound-rendering process onto another machine.

In the future, libCollider might be able to provide variable language bindings comparable to those of libpd, however, similar client libraries in other languages already exist, for example jCollider [12].

6. LIBCOLLIDER MULTI-LEVEL API

6.1. Low-level API - The Server Proxy

Since scsynth has a clearly documented set of the commands it understands and replies it returns [6], the initial motivation behind the API was to create one or more classes that handle sending those commands and properly dispatching any replies from the server. The initial result is the low level API contained in the class SCServer. SCServer has a full featured set of public members that currently include almost all commands found in the scsynth command reference [6]. SCServer also serves as a client-side proxy of a remote server instance. Thus, in a project you would typically have as many SCServer instances as you have server instances. SCServer conveniently abstracts all of the network and OSC code that is used to send and dispatch messages to and from the server via UDP or TCP. A code example of the low-level API looks like so:

6.2. Mid-level API - Quasi-SCLang

The motivation behind the middle portion of the API is to provide a set of proxy classes that mirror the structures involved in audio synthesis one works with when using scsynth. These include the familiar objects Buffer, But, Node and its subclasses, Synth and Group. These proxy classes provide a means to instantiate and retain a handle on server-side instances of these objects. Most of these classes are passed a pointer to a valid SCServer instance. The SCServer instance conveniently provides the necessary commands to instantiate and control the objects that wrap a pointer to it. An obvious benefit of this model is that each proxy class instance can leverage the same single SCServer instance for their instantiation and control, and perhaps most importantly, it can enable synchronization between the client application and server. For example, by instantiating a Buffer and calling one of its methods to load an arbitrary soundfile, the Buffer can use the SCServer for the low-level work of sending commands to the server, and in turn, populate the Buffer’s members such as channel count and sampling rate based off the reply from the server. This synchronization is important to the client in cases when channel count or sampling rate of the file determine what module is loaded or what event happens next in the client application, or perhaps when another class uses that information during initialization as we will see in the high-level portion of the API. An example of the mid-level API looks like so:

6.3. High-level API - Sound made easy

The most recent addition to the library, the Sound class, represents the start of the high-level portion of the API. This class is intended to be as intuitive and easy to use as possible, targeted towards developers with casual exposure to audio practices. The Sound class and future high-level classes are intended to make it easy to build and control one or many instances of simple to complex audio generators. These high-level classes can build directly on the classes from the low and mid-level portions of the API. For example, the Sound class represents a standard soundfile player, with looping, tape-head transposition, n-channel playback, seeking, gain control, fade in/out envelopes, and of course typical stop/start playback. In the simplest case, it uses internal Synth and Buffer members to load the necessary soundfile and control it via a synth definition on the server. The Sound class can be used for simple soundfile playback like so:

7. CALVR PROJECTS USING LIBCOLLIDER

We are working on a series of example applications, some of which leverage a simple generalized use of spatialized sound projection. Our specific interests in the use of libCollider can best be understood from the perspective of two projects.

7.1. Seismic Viewer: Increasing spatio-temporal resolution through audio-visual syncretism

While the visual sense is arguably superior when it comes to a precise spatial analysis of our surroundings, the auditory sense appears to be of a significantly higher temporal resolution. An integration of both is especially valuable as each sense can contribute its own potentials and strengths in the fused emergence of audio-visual syncretism. We are working on a series of example applications, some of which leverage a simple generalized use of spatialized sound projection. Our specific interests in the use of libCollider can best be understood from the perspective of two projects.

7.2. Virtual acoustic simulator

Another application of libCollider in the context of CalVR is the creation of a real-time architecture design tool that includes the rendering of an auditory impression of the resulting spatial structure. While the computational efficiency of scsynth is already beneficial, the inherently dynamic and scalable nature of scsynth’s rendering engine allows for increased navigability and support for more and more complex simulated geometries. A more detailed look at the virtual acoustic simulator will be the topic of a future publication.

8. CONCLUSION

While our own applications for libCollider are dominantly in the field of immersive visualization and audio-visual display, we hope that the application architecture of libCollider, both the separation of auditory and visual rendering into different threads controlled from a single C++ application as well as the use of SuperCollider’s powerful scsynth audio rendering system will prove to be a successful model to integrate audio into a wide variety of cross-platform applications—and that the multi-level API will be attractive to programmers of different interests and specialization levels. We are looking forward for further development and lively contact with other potential users and co-developers around the globe [3].

9. ACKNOWLEDGEMENTS

We would like to thank our collaborators from the Immersive Visualization Laboratory and the CISA3 workgroup at the Qualcomm Institute, namely Cathleen Hughes, David Strour, and Christopher McFarland, and we thank the King Abdullah City for Science and Technology, Saudi Arabia, for their partial funding of these developments. This work was developed in the Sonic Arts R&D group under the direction of Peter Otto. We are grateful for his continuing engagement, inspiration and support.

10. REFERENCES


of brightness with a stream of localized percussive sounds that represent a realization of global earthquake events presented in temporal scaling. Auditory events are scheduled at every CalVR frame, but a much higher resolution is achieved by using scsynth’s potential for a micro-delayed execution of the transmitted commands.
6.2. Mid-level API - Quasi-SCLang

The motivation behind the middle portion of the API is to provide a set of proxy classes that mirror the structures involved in audio synthesis one works with when using scsynth. These include the familiar objects Buffer, Bus, Node and its subclasses, Synth and Group. These proxy classes provide a means to instantiate and retain a handle on server-side instances of these objects. Most of these classes are passed a pointer to a valid SCServer instance. The SCServer instance conveniently provides the necessary commands to instantiate and control the objects that wrap a pointer to it. An obvious benefit of this model is that each proxy class instance can leverage the same single SCServer instance for their instantiation and control, and perhaps most importantly, it can enable synchronization between the client application and server. For example, by instantiating a Buffer and calling one of its methods to load an arbitrary soundfile, the Buffer can use the SCServer for the low-level work of sending commands to the server, and in turn, populate the Buffer’s members such as channel count and sampling rate based off the reply from the server. This synchronization is important to the client in cases when channel count or sampling rate of the file determine what module is loaded or what event happens next in the client application, or perhaps when another class uses that information during initialization as we will see in the high-level portion of the API. An example of the mid-level API looks like so:

6.3. High-level API - Sound made easy

The most recent addition to the library, the Sound class, represents the start of the high-level portion of the API. This class is intended to be as intuitive and easy to use as possible, targeted towards developers with casual exposure to audio practices. The Sound class and future high-level classes are intended to make it easy to build and control one or many instances of simple to complex audio generators. These high-level classes can build directly on the classes from the low and mid-level portions of the API. For example, the Sound class represents a standard soundfile player, with looping, tape-head transposition, n-channel playback, seeking, gain control, fade in/out envelopes, and of course typical stop/start playback. In the simplest case, it uses internal Synth and Buffer members to load the necessary soundfile and control it via a synth definition on the server. The Sound class can be used for simple soundfile playback like so:

7. CALVR PROJECTS USING LIBCOLLIDER

We are working on a series of example applications, some of which is a simple generalized use of spatialized sound projection. Our specific interests in the use of libCollider can best be understood from the perspective of two projects.

7.1. Seismic Viewer: Increasing spatio-temporal resolution through audio-visual syncretism

While the visual sense is arguably superior when it comes to a precise spatial analysis of our surroundings, the auditory sense appears to be of a significantly higher temporal resolution. An integration of both is especially valuable as each sense can contribute its own potentials and strengths in the fused emergence of audio-visual systems. We are working on a series of applications, some of which are a simple generalized use of spatialized sound projection. Our specific interests in the use of libCollider can best be understood from the perspective of two projects.
CONCERT SCOPE HEADPHONES

Masatoshi Hamanaka
University of Tsukuba
hamanaka@iit.tsukuba.ac.jp

Seunghoo Lee
University of Tsukuba
lee@kamei.tsukuba.ac.jp

ABSTRACT

We designed concert scope headphones that are equipped with a projector, an inclination sensor on the top of the headphones, and a distance sensor on the outside right headphone. We previously developed sound scope headphones that enable users to change the sound mixing depending on their head direction. However, the system could not handle images. In contrast, our headphones let the user listening to and watching a music scene scope on a particular part that he or she wants to hear and see. For example, when listening to jazz, one might want to more clearly hear and see the guitar or sax being played. The user can hear the guitar or sax sound from a frontal position by moving their head to the left or right. The user can adjust the distance sensor on the headphones and focus on a particular part they want to hear and see by simply putting their hand behind their ear.

1. INTRODUCTION

Our goal is to create an audio-visual interface that enables for the separation of the listening and watching of each performance if the user wants to clearly hear and see a particular performer. A musical expert at a concert, such as a conductor, can distinguish between the sounds coming from each performer, even if there are many performers playing the same instrument part. However, it is hard for musical novices to distinguish each performer’s sound. Therefore, we have developed concert scope headphones that let a user listening and watching a music performance to scope on a particular part of the performance if he or she wants to hear and see it.

We had to define two requirements for the interface. First, the user’s head direction is detected using inertial sensors. Next, the portion of the wide-angle image that captures the whole stage and smaller, we were able to develop a headphone device equipped with a compact projector, an inclination sensor, and a distance sensor (Figure 1). This device detects the user’s head direction, detects the distance between the user’s cupped hand and ear, and outputs the corresponding image and sound. Moreover, it is small enough to be used in the home as well as many other environments.

Figure 2 shows the system flow of our concert viewing headphones. First, the user’s head direction is detected using inertial sensors. Next, the portion of the wide-angle image that captures the whole stage corresponding to the head direction is extracted, and this portion is projected on the screen. At the same time, the recorded sounds are mixed to emphasize the sounds of the performers within the extracted portion (i.e., the projected image). If the user cups a hand to his or her ear to hear better, the projected image is enlarged to a degree corresponding to the distance between the user’s hand and the distance sensor attached to one of the headphones, enabling the user to better focus on a particular performer.

The concert scope headphones have three particular features.

Figure 1. Concert scope headphones.

Figure 2. System flow of audiovisual interface.