THE MOBILE CSOUND PLATFORM

Victor Lazzarini, Steven Yi, Joseph Timoney
Sound and Digital Music Technology Group
National University of Ireland, Maynooth, Ireland
victor.lazzarini@nuim.ie
stevenyi@gmail.com
jtimoney@cs.nuim.ie

Damian Keller
Nucleo Amazonico de Pesquisa Musical (NAP)
Universidade Federal do Acre, Brazil
dkeller@ccrma.stanford.edu

Marcelo Pimenta
LCM, Instituto de Informatica
Universidade Federal do Rio Grande do Sul, Brazil
mpimenta@info.ufrgs.br

ABSTRACT

This article discusses the development of the Mobile Csound Platform (MCP), a group of related projects that aim to provide support for sound synthesis and processing under various new environments. Csound is itself an established computer music system, derived from the MUSIC N paradigm, which allows various uses and applications through its Application Programming Interface (API). In the article, we discuss these uses and introduce the three environments under which the MCP is being run. The projects designed for mobile operating systems, iOS and Android, are discussed from a technical point of view, exploring the development of the CsoundObj toolkit, which is built on top of the Csound host API. In addition to these, we also discuss a web deployment solution, which allows for Csound applications on desktop operating systems without prior installation. The article concludes with some notes on future developments.

1. INTRODUCTION

Csound is a well-established computer music system in the MUSIC N tradition [1] , developed originally at MIT and then adopted as a large community project, with its development base at the University of Bath. A major new version, Csound 5, was released in 2006, offering a completely re-engineered software, as a programming library with its own application programming interface (API).
This allowed the system to be embedded and integrated into many applications. Csound can interface with a variety of programming languages and environments (C/C++, Objective C, Python, Java, Lua, Pure Data, Lisp, etc.). Full control of Csound compilation and performance is provided by the API, as well software bus access to its control and audio signals, and hooks into various aspects of its internal data representation. Composition systems, signal processing applications and various frontends have been developed to take advantage of these features. The Csound API has been described in a number of articles [4], [6] [7].

New platforms for Computer Music have been brought to the fore by the increasing availability of mobile devices for computing (in the form of mobile phones, tablets and netbooks). With this, we have an ideal scenario for a variety of deployment possibilities for computer music systems. In fact, Csound has already been present as the sound engine for one of the pioneer portable systems, the XO-based computer used in the One Laptop per Child (OLPC) project [5]. The possibilities allowed by the reengineered Csound were partially exploited in this system. Its development sparked the ideas for a Ubiquitous Csound, which is now steadily coming to fruition with a number of parallel projects, collectively named the Mobile Csound Platform (MCP). In this paper, we would like to introduce these and discuss the implications and possibilities provided by them.

2. THE CSOUND APPLICATION ECOSYSTEM

Csound originated as a command-line application that parsed text files, setup a signal processing graph, and processed score events to render sound. In this mode, users hand-edit text files to compose, or use a mix of handedited text and text generated by external programs. Many applications–whether custom programs for individual use or publicly shared programs–were created that could generate text files for Csound usage. However, the usage scenarios were limited as applications could not communicate with Csound except by what they could put into the text files, prior to starting rendering.

Csound later developed realtime rendering and event input, with the latter primarily coming from MIDI or standard input, as Csound score statements were also able to be sent to realtime rendering Csound via pipes. These features allowed development of Csound-based music systems that could accept events in realtime at the note-level,

such as Cecilia [8]. These developments extended the use cases for Csound to realtime application development.

However, it was not until Csound 5 that a full API was developed and supported that could allow finer grain interaction with Csound [3]. Applications using the API could now directly access memory within Csound, control rendering frame by frame, as well as many other low-level features. It was at this time that desktop development of applications grew within the Csound community. It is also this level of development that Csound has been ported to mobile platforms.

Throughout these developments, the usage of the Csound language as well as exposure to users has changed as well. In the beginning, users were required to understand Csound syntax and coding to operate Csound. Today, applications are developed that expose varying degrees of Csound coding, from full knowledge of Csound required to none at all. Applications such as those created for the XO platform highlight where Csound was leveraged for its audio capabilities, while a task-focused interface was presented to the user. Other applications such as Cecilia show where users are primarily presented with a task-focused interface, but the capability to extend the system is available to those who know Csound coding. The Csound language then has grown as a means to express a musical work, to becoming a domain-specific language for audio engine programming.

Today, these developments have allowed many classes of applications to be created. With the move from desktop platforms to mobile platforms, the range of use cases that Csound can satisfy has achieved a new dimension.

3. CSOUND FOR IOS

At the outset of this project, it was clear that some modifications to the core system would be required for a full support of applications on mobile OSs. One of the first issues arising in the development of Csound for iOS was the question of plugin modules. Since the first release of Csound 5, the bulk of its unit generators (opcodes) were provided as dynamically-loaded libraries, which resided in a special location (the OPCODEDIR or OPCODEDIR64 directories) and were loaded by Csound at the orchestra compilation stage. However, due to the uncertain situation regarding dynamic libraries (not only in iOS but also in other mobile platforms), it was decided that all modules without any dependencies or licensing issues could be moved to the main Csound library code. This was a major change (in Csound 5.15), which made the majority of opcodes part of the base system, about 1,500 of them, with the remaining 400 or so being left in plugin modules. The present release of Csound for iOS includes only the internal unit generators.

With a Csound library binary for iOS (in the required arm and x86 architectures, for devices and simulators), a new API was created in Objective-C, called CsoundObj. This is a toolkit that provides a wrapper around the standard Csound C API and manages all hardware connectivity. A CsoundObj object controls Csound performance and provides the audio input and output functionality, via the CoreAudio AuHAL mechanism. MIDI input is also handled either by the object, by allowing direct pass-through to Csound for standard Csound MIDI-handling, or by routing MIDI through a separate MIDIManager class to UI widgets, which in turn send values to Csound. Additionally, a number of sensors that are found on iOS devices come pre-wrapped and ready to use with Csound through CsoundObj.

To communicate with Csound, an object-oriented callback system was implemented in the CsoundObj API. Objects that are interested in communicating values, whether control data or audio signals, to and from Csound must implement the CsoundValueCacheable protocol. These CsoundValueCacheables are then added to CsoundObj and values will then be read from and written to on each control cycle of performance (fig.1). The CsoundObj API comes with a number of CsoundValueCacheables that wrap hardware sensors as well as UI widgets, and examples of creating custom CsoundValueCacheables accompany the Csound for iOS Examples project.

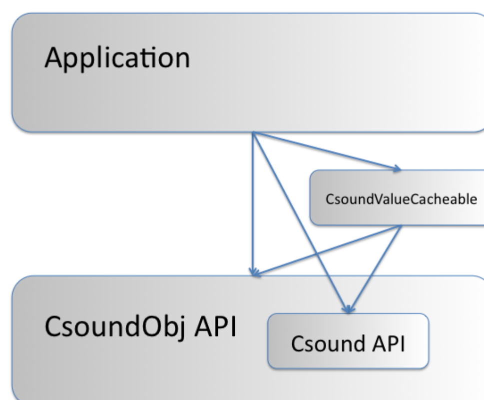

Figure 1. CsoundObj and the Application

While the CsoundObj API covers most of the general use cases for Csound, it does not wrap the Csound C API in its entirety. Instead, the decision was made to handle the most common use cases from Objective-C, and for less used functions, allow retrieval of the CSOUND object. This is the lower-level object that encapsulates all of the C API functionality. It is a member of CsoundObj and it is exposed so that developers can use methods not directly available in that class. It is expected that as more developers use CsoundObj, the CsoundObj API may continue to further wrap C API functions as they are identified as being popular.

Together with the API for iOS, a number of application examples complete the SDK. These can be used during development both as a practical guide for those interested in using Csound on iOS, as well as a test suite for the API. Examples include a number of realtime instruments

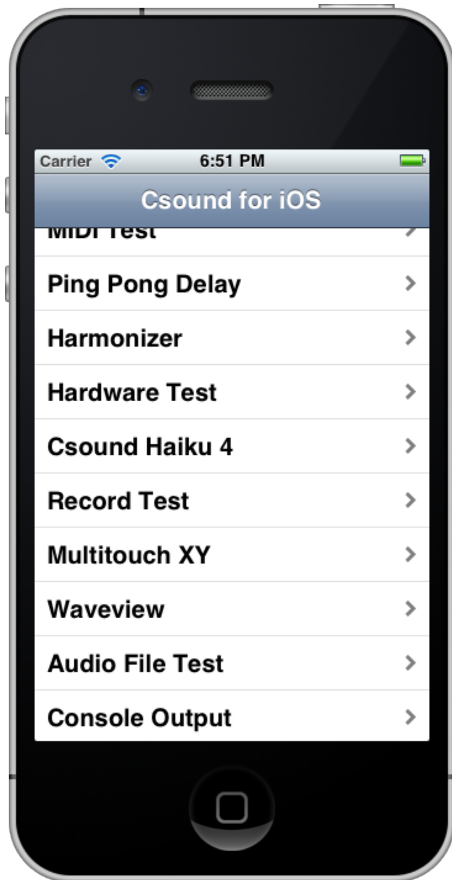

Figure 2. Csound for iOS SDK sample app

(performed by screen or MIDI input), signal processing applications (harmonizer, pitch shifter, ping-pong echo), a generative music example, and other audio-related utilities (fig.2). These examples, together with the manual created for the project, were assembled to assist in learning Csound for iOS.

4. CSOUND FOR ANDROID

Csound for Android is based on a native shared library (libcsoundandroid.so) built using the Android Native Development Kit (NDK)[1], as well as pure Java code for the Android Dalvik compiler. The native library is composed by the object files that are normally used to make up the main Csound library (libcsound), its interfaces extensions (libcsnd), and the external dependency, libsndfile[2]. The Java classes include those commonly found in the csnd.jar library used in standard Java-based Csound development (which wrap libcsound and libcsnd), as well as unique classes created for easing Csound development on Android.

As a consequence of this, those users who are familiar with Csound and Java can transfer their knowledge when working on Android. Developers who learn Csound on Android can take their experience and work on standard Java desktop applications. The two versions of Java do differ, however, in some areas such as classes for accessing hardware and different user interface libraries. Similarly to iOS, in order to help ease development, a CsoundObj class, here written in Java, of course, was developed to provide straightforward solutions for common tasks.

As with iOS, some issues with the Android platform have motivated some internal changes to Csound. One such problem was related to difficulties in handling temporary files by the system. As Csound was dependent on these in the compilation/parsing stage, a modification to use core (memory) files instead of temporary disk files was required.

Two options have been developed for audio IO. The first involves using pure Java code through the AudioTrack API provided by the Android SDK. This is, at present, the standard way of accessing the DAC/ADC, as it appears to provide a slightly better performance on some devices. It employs the blocking mechanism given by AudioTrack to push audio frames to the Csound input buffer (spin) and to retrieve audio frames from the output buffer (spout), sending them to the system sound device. Although low latency is not available in Android, this mechanism works satisfactorily.

As a future low-latency option, we have also developed a native code audio interface. It employs the OpenSL API offered by the Android NDK. It is built as a replacement for the usual Csound IO modules (portaudio, alsa, jack, etc.), using the provided API hooks. It works asynchronously, integrated into the Csound performance cycle. Currently, OpenSL does not offer lower latency than AudioTrack, but this situation might change in the future, so this option has been maintained alongside the pure Java implementation. It is presented as an add-on to the native shared library. Such mechanism will also be used for the future addition of MIDI IO (replacing the portmidi, alsamidi, etc. modules available in the standard platforms), in a similar manner to the present iOS implementation.

At the outset of the development of Csound for Android, a choice was made to port the CsoundObj API from Objective-C to Java. The implementation of audio handling was done so in a manner following the general design as implemented on iOS (although, internally, the current implementations differ in that iOS employs an asynchronous mechanism, whereas in Android blocking IO is used). Also, the APIs match each other as much as possible, including class and method names. There were inevitable differences, resulting primarily from what hardware sensors were available and lack of a standard MIDI library on Android. However, the overall similarities in the APIs greatly simplified the porting of example applications from iOS to Android. For application developers using MCP, the parity in APIs means an easy migration path when moving projects from one platform to the other.

[1] http://developer.android.com/sdk/ndk/index.html
[2] http://www.mega-nerd.com/libsndfile/

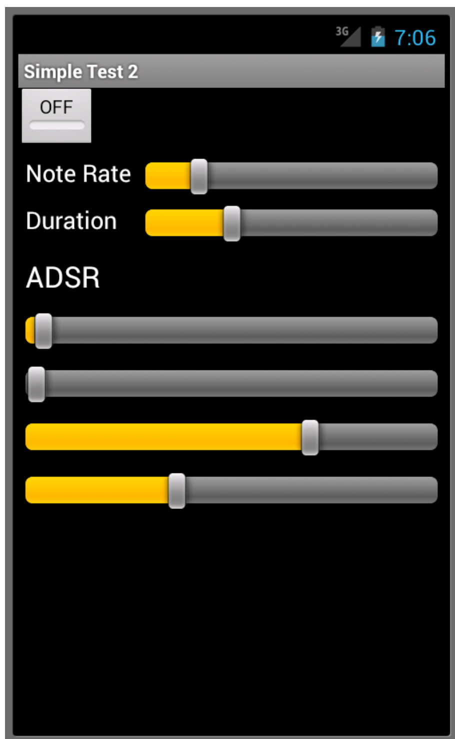

Figure 3. Csound for Android SDK example

5. CSOUND FOR JAVA WEB START

Csound 5 has long included a Java wrapper API that is used by desktop applications such as *AVSynthesis* and *blue*. During research for a music-related project that required being deployable over the web, work was done to explore using Java as the technology to handle the requirements of the project, particularly Java Web Start (JAWS). The key difference between ordinary Java desktop and Java Web Start-based applications is that with the former, the Csound library must be installed by the user for the program to function. With the latter, instead, the application will be deployed, downloading the necessary libraries to run Csound without the user having anything installed (besides the Java runtime and plugin).

Regarding security, JAWS allows for certificate-signed Java applications to package and use use native libraries. Typically, JAWS will run an application within a *sandbox* that limits what the application is allowed to do, including things like where files can be written and what data can be read from the user's computer. However, to run with native libraries, JAWS requires use of all permissions, which allows full access to the computer. Applications must still be signed, verifying the authenticity of what is downloaded, and users must still allow permission to run. This level of security was deemed practical and effective enough for the purposes of this research.

In order to keep the native library components to a minimum, JAWS Csound only requires the Csound core code (and soundfile access through libsndfile, which is packaged with it). Audio IO is provided by the Java-Sound library, which is a standard part of modern Java runtime environments. JAWS Csound has been chosen as the sound engine for the DSP eartraining online course being developed at the Norwegian University of Science and Technology [2].

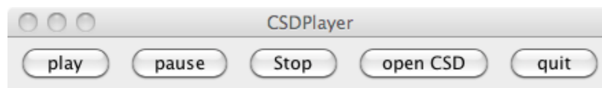

Figure 4. Csound for JAWS example

6. CSOUND 6

In February 2012, the final feature release of Csound 5 was launched (5.16) with the introduction of a new bison/flex-based orchestra parser as default. The development team has now embarked on the development of the next major upgrade of the system, Csound 6. The existence of projects such as the MCP will play an important part in informing these new developments. One of the goals for the new version is to provide more flexibility in the use of Csound as a synthesis engine by various applications. This is certainly going to be influenced by the experience with MCP. Major planned changes for the system will include:

- Separation of parsing and performance
- Loading/unloading of instrument definitions
- Further support for parallelisation

As Csound 6 is developed, it is likely that new versions of the MCP projects will be released, in tandem with changes in the system.

7. CONCLUSIONS

The Mobile Csound Platform has been developed to bring Csound to popular mobile device operating systems. Work was done to build an idiomatic, object-oriented API for both iOS and Android, implemented using their native languages (Objective-C and Java respectively). Work was also done to enable Csound-based applications to be deployed over the internet via Java Web Start. By porting Csound to these platforms, Csound as a whole has moved from embracing usage on the desktop to become pervasively available. The MCP, including all the source code for the SDK, and technical documentation, is available for download from

http://sourceforge.net/projects/csound/files/csound5

For the future, it is expected that current work on Csound 6 will help to open up more possibilities for music application development. Developments such as real-time orchestra modification within Csound should allow for more

flexibility in kinds of applications that are possible to develop. As mobile hardware continues to increase in number of cores and multimedia capabilities, Csound will continue to grow and support these developments as first-class platforms.

8. ACKNOWLEDGEMENTS

This research was partly funded by the Program of Research in Third Level Institutions (PRTLI 5) of the Higher Education Authority (HEA) of Ireland, through the Digital Arts and Humanities programme.

9. REFERENCES

[1] R. Boulanger, Ed., *The Csound Book*. Cambridge, Mass: MIT Press, 2000.

[2] O. Brandtsegg, J. Inderberg, H. Kvidal, V. Lazzarini, J. Rudi, S. Saue, A.Tidemann, N. Thelle, and J. Tro, “Developing an online course in dsp eartraining,” *submitted to DAFx 2012*, 2012.

[3] J. ffitch, “On the design of csound 5,” in *Proceedings of 4th Linux Audio Developers Conference*, Karlsruhe, Germany, 2006, pp. 79–85.

[4] V. Lazzarini, “Scripting csound 5,” in *Proceedings of 4th Linux Audio Developers Conference*, Karlsruhe, Germany, 2006, pp. 73–78.

[5] ——, “A toolkit for audio and music applications in the xo computer,” in *Proc. of the International Computer Music Conference 2008*, Belfast, Northern Ireland, 2008, pp. 62–65.

[6] V. Lazzarini and J. Piche, “Cecilia and tclcsound,” in *Proc. of the 9th Int. Conf. on Digital Audio Effects (DAFX)*, Montreal, Canada, 2006, pp. 315–318.

[7] V. Lazzarini and R. Walsh, “Developing ladspa plugins with csound,” in *Proceedings of 5th Linux Audio Developers Conference*, Berlin, Germany, 2007, pp. 30–36.

[8] J. Piche and A. Burton, “Cecilia: a production interface for csound,” *Computer Music Journal*, vol. 22, no. 2, pp. 52–55, 1998.