DIGITAL AUDIO EFFECTS ON MOBILE PLATFORMS

Victor Lazzarini, Steven Yi, and Joseph Timoney

Sound and Digital Music Technology Group
National University of Ireland, Maynooth
victor.lazzarini@nuim.ie,
stevenyi@gmail.com,
joseph.timoney@nuim.ie

ABSTRACT

This paper discusses the development of digital audio effect applications in mobile platforms. It introduces the Mobile Csound Platform (MCP) as an agile development kit for audio programming in such environments. The paper starts by exploring the basic technology employed: the Csound Application Programming Interface (API), the target systems (iOS and Android) and their support for realtime audio. CsoundObj, the fundamental class in the MCP toolkit is introduced and explored in some detail. This is followed by a discussion of its implementation in Objective-C for iOS and Java for Android. A number of application scenarios are explored and the paper concludes with a general discussion of the technology and its potential impact for audio effects development.

1. INTRODUCTION

Lately, substantial interest has been directed at the possibilities offered by mobile platforms, such as the ones running iOS (iPod Touch, iPhone and iPad) and Android (mobile phone and tablets from various vendors) operating systems. One of the major features of these systems is the potential for media applications, in which digital audio effects have an important part to play. In such a scenario, one key aspect for programmers is the presence of a number of rapid development tools. These facilitate the task of creating applications for a variety of uses.

In this paper, we would like to introduce a software development kit that allows for rapid development of audio applications on mobile systems, the Mobile Csound Platform (MCP). It allows the application of a mature sound processing technology, the Csound language and engine, to a variety of ends. This paper is organised as follows: we will first discuss the technology on which MCP is based and its target platforms. Following this we will discuss the detail of the iOS and Android implementations, with examples for each platform. The paper concludes with an overview of applications and a discussion of the technologies employed.

2. TECHNOLOGY INFRASTRUCTURE

In this section, we discuss the technology infrastructure on which MCP is build: Csound and its target systems (iOS, Android). The computing devices used in these platforms is quite varied, but they have some commonalities, and they have been in constant improvement. Most of the devices will be based on the ARM processor, of various versions. Earlier Android phones and tablets will have used the ARM v.5, which does not include a floating-point coprocessor. On these, audio processing applications will have a weak performance. However, newer models have excellent computing capabilities, including multiple cores and fast clock rates.

2.1. Csound

Csound is a a mature computer music language based on the MUSIC N paradigm [1], which was developed originally at MIT by Barry Vercoe and then expanded as part of a community project, managed by John ffitch, with a number of collaborators. Csound 5 [2], released in 2006, was a major re-engineering of the software, which became a programming library with an application programming interface (API). This allowed embedding and integration of the system with a variety of 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 [3], [4] [5].

The employment of Csound in mobile audio applications is a natural extension of its use cases. 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 [6]. The possibilities allowed by the re-engineered Csound were partially exploited in this system. Its development sparked the ideas for a Ubiquitous Csound, which is now steadily coming to fruition with MCP.

Csound 5 is composed of its main C-based API (which provides access to libcsound) and an auxiliary 'interfaces' library (libcsnd), which has a C++ API. It has a single dependency on the libsndfile library, which either must be present on the target system or needs to supplied alongside Csound. On iOS, both Csound and libsndfile are built as static libraries that are linked against by the main application, and the interfaces library is not used. On Android, Csound and libcsnd are built into a single dynamic library, which is statically linked against libsndfile.

Since the first release of Csound 5, the bulk of its unit generators (opcodes) has been provided in dynamic libraries, loaded at the orchestra compilation stage.To facilitate the development of MCP, all modules without any dependencies or licensing issues were moved to the main Csound library. 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 MCP includes only the internal opcodes.

2.2. Audio and MIDI on iOS

The audio subsystem on iOS offers developers a subset of the original CoreAudio framework, which is shipped with Apple's desktop operating system, OSX. In that, it does not allow the simple porting of desktop audio applications, as some of the underlying code might not be supported. In particular, the HAL (Hardware Audio Layer) of CoreAudio is not present, but a wrapper, in the form of an AudioUnit, AuHAL, is used instead. In general, OSX applications using the AuHAL AudioUnit could be easily ported to iOS, as far as the audio code is concerned. While in general applications are written in Objective-C, CoreAudio can be also accessed via C-language code, which can be called directly by an Objective-C class.

The AuHAL API does not allow the direct setting of audio buffer sizes. Instead, it will take hints regarding the desired size, but this is not guaranteed to be provided. In this case, the latency is effectively fixed by the system, and software clients are not offered the option of modifying it. On iOS 5, under an iPhone 3G and iPad 1, buffer size was determined to be 512 frames, which at 44100 samples/sec, provides about 11.6 ms of latency. While this does not qualify by any means as low-latency, it allows for a number of audio processing applications.

The latest versions of iOS also include support for MIDI via the CoreMidi framework, which includes most of the features found in its OSX version. While MIDI hardware is not provided on the devices, on the iPad, there is the option of USB MIDI devices, via an adaptor to its connector port, in addition to network MIDI communications. In general, iOS provides a good infrastructure for audio development and it has been the main target for sound/media software vendors.

Deployment of iOS applications, however, is not straightforward. While it is possible to upload applications to devices connected to a development computer running OSX (with a developer's license), for a more wide distribution of applications, the only form is via the Apple Store. This imposes a number of restrictions. For instance, the approval status by Apple of some open-source licenses is not very clear. Applications need to go through a vetting process, which can be long. So while the development can be classed as rapid, deployment is nothing of the kind.

2.3. Audio on Android

Android is an open-source operating system based on the Linux kernel. Its software development kit (SDK) is based on the Java language, although it does not support the original Java Runtime Environment. Instead, Android provides its own implementation, which runs on the Dalvik Virtual Machine. In addition to the SDK, the system also supports a native development kit (NDK), which allows code written in C or C++ to be built into dynamic libraries. This code can then be accessed by applications via the Java Native Interface (JNI), which supports the loading of dynamic modules.

Audio on Android has been in constant development since the first versions of the system. Prior to Android-9 (version 2.3.1), the option for streaming audio input and output was provided by a Java-based API, AudioTrack. Processing code could be written in pure Java or as C/C++ libraries (as discussed above). From Android-9 onwards, the system offers programmers a NDK alternative, based on the OpenSL ES standard of Khronos Computing. Audio applications can access the audio device directly using C-language code, outside the virtual machine (and its garbage collector), which, from the computing side, is optimal. In that case, the application code in Java can then be left to control tasks and the whole of the audio code can be placed in native module.

The 'elephant in the room' of audio development on Android is its lack of support for reasonable latencies. At the moment, this appears to be device dependent, but ranging from 200ms at the shortest to over 500ms. Regardless of the API (AudioTrack or OpenSL), the observed latency is the very same. This is in fact determined by the common underlying audio library, AudioFlinger. Since Android does not include a full client-side Alsa library (libasound), it is not possible to employ any of the well-known methods developed for GNU/Linux operating systems. It is hoped that this situation will be resolved in the near future.

On the bright side, Android is a much more open system than iOS, and is supported widely. Deployment of applications is very much simplified: an apk package (the Android application format) can be downloaded and installed directly from a given URL. Development tools for all major OSs are available, so programmers are not tied down to a particular system or even to a particular integrated development environment (IDE). With the latency issue solved, Android will be potentially a primary platform for open-source audio applications.

3. THE MCP TOOLKIT

3.1. CsoundObj

At the heart of the MCP toolkit is based on the CsoundObj class (with its different Objective-C and Java incarnations). This is responsible for encapsulating the Csound API calls and incorporating a control-data mechanism for passing values from the application to Csound, as well as other out-of-the-box solutions for common tasks (such as routing audio from Csound to hardware output). Figure 1 shows how CsoundObj, Csound and the application are integrated in MCP.

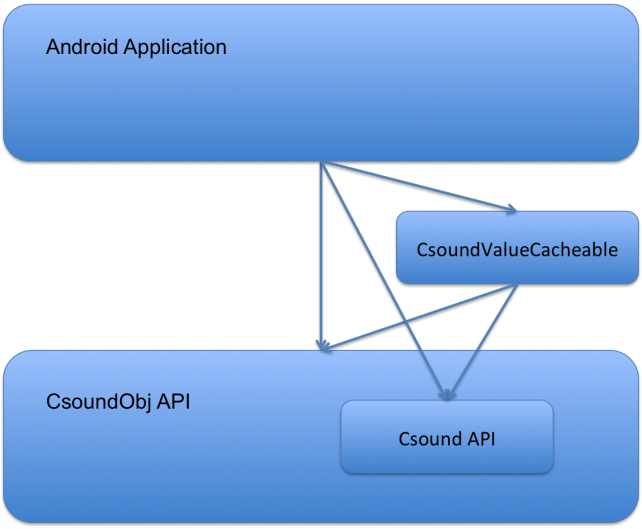

Figure 1: *CsoundObj and the Application*

The essential elements of the CsoundObj class are defined (in pseudocode) as:

```
class CsoundObj{
// the csound instance
Csound csound;
 // ValueCacheable list
List<CsoundValueCacheable> valuesCache;
// CompletionListener list
List<CsoundCompletionListener>
         completionListener;

public:
CsoundObj(); // constructor
Csound getCsound(); // get Csound instance

// ValueCacheable list manipulation
addValueCacheable(CsoundValueCacheable
        valueCacheble);
removeValueCacheable(CsoundValueCacheable
         valueCacheble);

// Completion Listener list manipulation
addCompletionListener(
             CsoundCompletionListener
             completionListener);

// interface components
CsoundValueCacheable
   addSlider(Slider slider, String channel);
CsoundValueCacheable
   addButton(Button button, String channel);
... // other supported UI components

// hardware
CsoundValueCacheable enableAccelerometer();
... // other supported HW components

// Score interface for RT events
sendScore(String score);

// csound attributes
getNumChannels();
getKsmps();

// Performance control
startCsound(File csdfile);
stopCsound();
muteCsound();
unmuteCsound();
}
```

CsoundObj contains a Csound instance and two lists, of CsoundValueCacheable and CsoundCompletionListener objects. The former encapsulates the synthesis engine, while the latter are responsible for the communication between the device controls and Csound. For each interface component, a CsoundValueCacheable acts as an intermediary to update values from and to Csound, working at block boundaries to enable atomic operations:

```
class CsoundValueCacheable {
public:
setup(CsoundObj csoundObj);
updateValuesToCsound();
updateValuesFromCsound();
cleanup();
}
```

It employs the Csound software bus through named channels. For instance, when adding a UI element to the application, a bus channel is defined and this can be used in Csound to retrieve the data:

```
kvar chnget "channel_name"
```

Methods for manipulating the data members of CsoundObj are provided: for adding UI components, HW controls, starting/stopping the engine, sending events to it, etc.. 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 or Java, and for less used functions, allow retrieval of the Csound object. This is exposed so that developers can use methods not directly available in that class. It is expected that as more developers use CsoundObj, this class might continue to further wrap C API functions as they are identified as being commonly used.

Some small implementation differences remain, as the SDK and hardware devices are different, but the overall principles outlined above are present in both iOS and Android. Applications using CsoundObj should benefit from a certain amount of portability from one platform to another.

3.2. Csound for iOS

The Csound for iOS platform is based on a static libcsound.a and the CsoundObj class implementation in Objective-C, as well as the dependency libsndfile.a. The Objective-C class has all the attributes and methods presented above, plus some extra support for gyroscope and attitude. The class is derived from the base class NSObject and the Csound object is actually wrapped in a C structure (csdata), which is used to facilitate the interface between the C-language CoreAudio and the application code:

```
@interface CsoundObj : NSObject {
   csdata mCsData; // Csound data
   NSMutableArray* valuesCache;
   NSMutableArray* completionListeners;
   BOOL mMidiInEnabled; // app midi
   CMMotionManager* mMotionManager;
   NSURL *outputURL; // audio outfile
   SEL mMessageCallback; // console msgs
   id mMessageListener;
}
```

In addition to the basic startCsound() method as described above, we have an alternative method to record its output to a file (given by a URL), which can also be enabled after Csound starts by another method:

```
-(void)startCsound:(NSString*)csdFilePath;
-(void)startCsound:(NSString *)csdFilePath
        recordToURL:(NSURL *)outputURL;
-(void)recordToURL:(NSURL *)outputURL;
```

As mentioned before, audio implementation uses CoreAudio, which is a callback-based API. Csound's performKsmps() function, which processes one block of audio, is called inside the CoreAudio callback to consume and fill the input and output buffers, respectively. Control values held by CsoundValueCacheable() are passed to and updated by Csound at audio block boundaries in the callback processing loop.

MIDI is implemented at two levels: using the MIDI IO setup functions of the Csound API (in C); and at application-level, integrated to UI widgets (in Objective-C). This allows developers to use all the standard MIDI functionality (via the Csound opcodes) and/or apply MIDI to the controls, which will indirectly be connected to Csound synthesis code.

A simple example of how to setup and start Csound a performance on iOS is shown below. Here csound is a CsoundObj object, which gets constructed and added a slider linked to a channel ("slider"). It is then started by passing it a file containing Csound code:

```
NSString *tempFile =
   [[NSBundle mainBundle]
      pathForResource:@"test"
      ofType:@"csd"];
self.csound = [[CsoundObj alloc] init];
[self.csound addCompletionListener:self];
[self.csound addSlider:mSlider
          forChannelName:@"slider"];
[self.csound startCsound:tempFile];
```

A minimal Csound code that could be employed by this example would look as follows:

```
<CsoundSynthesizer>
<CsOptions>
-o dac -+rtaudio=null -d
</CsOptions>
<CsInstruments>
ksmps=64

instr 1
ksl chnget "slider"
ksl port ksl, 0.01
a2 madsr 0.01,0.1,0.8,0.1
a1 oscili a2*0dbfs, p5*(1+ksl), 1
out a1
endin

</CsInstruments>
<CsScore>
f0 3600
f1 0 16384 10 1
i1 -1 0.5 440
</CsScore>
</CsoundSynthesizer>
```

3.3. Csound for Android

The Csound for Android platform is made up of a native shared library (libCsoundandroid.so) built using the NDK, as well as Java classes compliant the Android Dalvik implementation. The Java classes include those commonly found in the csnd.jar library used for desktop Java-based Csound development, as well as unique classes created for easing Csound development on Android.

The SWIG[1] wrapping used for Android contains all of the same classes as those used in the Java wrapping that is used for desktop Java development with Csound. Consequently, those users who are familiar with Csound and Java can transfer their knowledge when working on Android, and users who learn Csound development on Android can take their experience and work on desktop Java applications. However, the two platforms do differ in some areas such as classes for accessing hardware and different user interface libraries.

The CsoundObj Java implementation follows much of the specification described above:

```
public class CsoundObj {
private Csound csound;
private ArrayList<CsoundValueCacheable>
        valuesCache;
private ArrayList<CsoundObjCompletionListener>
        completionListeners;
private boolean muted = false;
private boolean stopped = false;
private Thread thread;
...
};
```

The internal implementation of CsoundObj differs somewhat to the iOS version, mostly due to the specifics of the audio APIs in Android. Here, a Thread object is used to spawn a separate processing thread, where calls to Csound.PerformKsmps() will be made. Audio IO has been developed in two fronts. The first is as a Csound realtime audio IO module employing the OpenSL API, which is offered by the Android NDK. This is built as a straight replacement for the usual Csound IO modules (PortAudio, ALSA, JACK, etc.), using the provided API hooks.

In this case, the Csound input and output functions, called synchronously in its performance loop, pass a buffer of audio samples to the DAC/ADC using the OpenSL enqueue mechanism. The OpenSL module is the default mode of IO in Csound for Android. Although it does not currently offer low-latency, it is a more efficient means of passing data to the audio device and it operates outside the influence of the Dalvik virtual machine garbage collector (which executes the Java application code).

Alternatively, a pure Java solution is also provided through the AudioTrack API offered by the Android SDK The AudioTrack code offers an alternative means accessing the device. It pushes/retrieves input/output frames into/from the main processing buffers (spin/spout) of Csound synchronously at control cycle intervals (in the Java processing thread). It is offered as an option to developers, which can be used for instance, in older versions of Android without OpenSL support.

An example of the use of CsoundObj in Android is shown below, as an exact port of the iOS example above. A resource is obtained, which contains the Csound code, this is passed to Csound as it is started. The addSlider() method is also used to add a slider control:

```
String csd =
 getResourceFileAsString(R.raw.test);
File f = createTempFile(csd);
csoundObj.addSlider(fSlider,
       "slider", 0.0, 1.0);
csoundObj.startCsound(f);
```

4. APPLICATIONS

While Csound is both directed at both users and developers, MCP is aimed primarily at application programmers. It serves as the audio component in an agile development environment, alongside other elements. The audio synthesis and processing aspect of applications can be quickly prototyped using one of the Csound desktop frontends and then transported into the application.

As an example of this approach, the MCP SDK release includes a number of applications, which are designed to highlight the various types of synthesis and digital audio effects that are possible under it (figs. 2 and 3):

1. Simple Test 1: demonstrates an arpeggio pattern whose pitches are controllable via a slider.
2. Simple Test 2: a generative example with ADSR, note rate and duration controls.

[1] http://www.swig.org

3. Button Test: shows the use of buttons as event triggers and general-purpose controls.
4. MIDI test: a polyphonic MIDI subtractive synthesizer, with ADSR and filter controls (iOS only).
5. Ping-pong delay: a processing example using a stereo delayline.
6. Harmonizer: a spectral processing example implementing a formant-preserving harmonizer.
7. Hardware test: demonstrates the accelerator controlling an oscillator and filter.
8. Csound Haiku 4: an example of a generative composition by Iain McCurdy.
9. Multitouch XY: demonstrates a 2-D interface with multitouch capabilities controlling a polyphonic synthesizer.

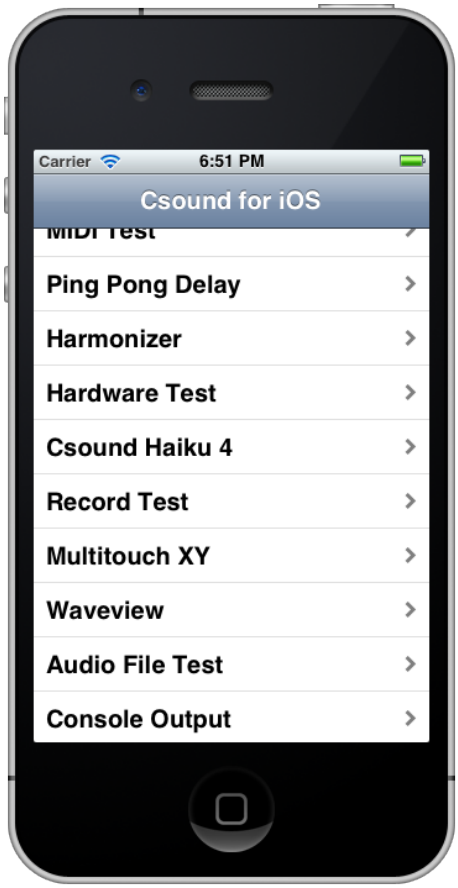

Figure 2: *Csound for iOS SDK application examples*

For audio signal processing research, the environment can be used as a means of developing portable demonstrations and prototypes (see the discussion section below). For commercial application developers, it should provide a simple to use audio engine. In fact, at the time of writing, a number of commercial applications for iOS have been developed and will shortly be available.

5. DISCUSSION

5.1. The multi-language design paradigm

At the core of application programming using MCP lies a multi-language design paradigm, that of the use of an application-level language (Java or ObjectiveC), a system-implementation language (C/C++) and a domain-specific language (DSL, in this case, Csound). This arrangement is very similar to what has been advanced by authors such as Ousterhout [7], where glue code is used to connect components implemented in languages that are closer to hardware. Such glue code in this case is both the application code (which manages the UI, interaction and controls the audio computation) and the audio-specific code (which connects unit generators, etc., and sets up a synthesis engine). It also demonstrates the principle of separation of concerns in software design, where a complex problem (in this case the design of an audio-processing application) is divided into a set of discrete parts, that are implemented separately [8].

This approach is not particularly new. In fact, it has been present in Computer Music since MUSIC IV, where the event generation for a given score was completely separated from the design of the synthesis program which would receive the score [9]. In fact the MUSIC N languages themselves can be seen as glue code connecting unit generators implemented in a lower-level language [10]. This concept was extended to be the basis for the CARL system, where components where connected using shell scripting [11]. Similarly, it was used in Cmix [12], where synthesis programs written in C were glued by a score code written in MinC. More recently, it is the basis of the meta-programming principles embodied by Faust [13]. Since the advent of Csound 5, it has become an organic mode of operation, where Csound can be part of a system involving a number of other languages. In fact, Csound itself can embed other languages (such as Python and Lua).

This approach is, however, not free of criticism. Single-language proponents advocate that even for specific tasks such as audio and music programming, there should be a single language responsible for all aspects of an application, from audio code to UI to program management. But in fact, even systems that have invested heavily in such approach might depend on auxiliary languages for some types of application development.

Our contention is that the multi-language paradigm offers significant benefits to the task of audio application development. A DSL such as Csound is well developed and maintained, as well as comprehensive. It is also completely backwards compatible, so it can take advantage of a library of code that has been developed over more than thirty years (MUSIC11 code can in most cases run in Csound). It allows, with MCP, audio programming to be fully cross-platform, from iOS to Android and to desktop operating systems.

For the task of translating algorithms, Csound is capable of representing signal flowcharts in a very straightforward and readable way, which can be a problem in some cases with Faust and with graphic flowchart languages such as Pure Data (Pd) [14]. Also, it allows simple access to sample-level computation, which, while not completely impossible, is awkward in Pd. Multirate processing can also be implemented and spectral-domain data manipulation is one of its strengths [15].

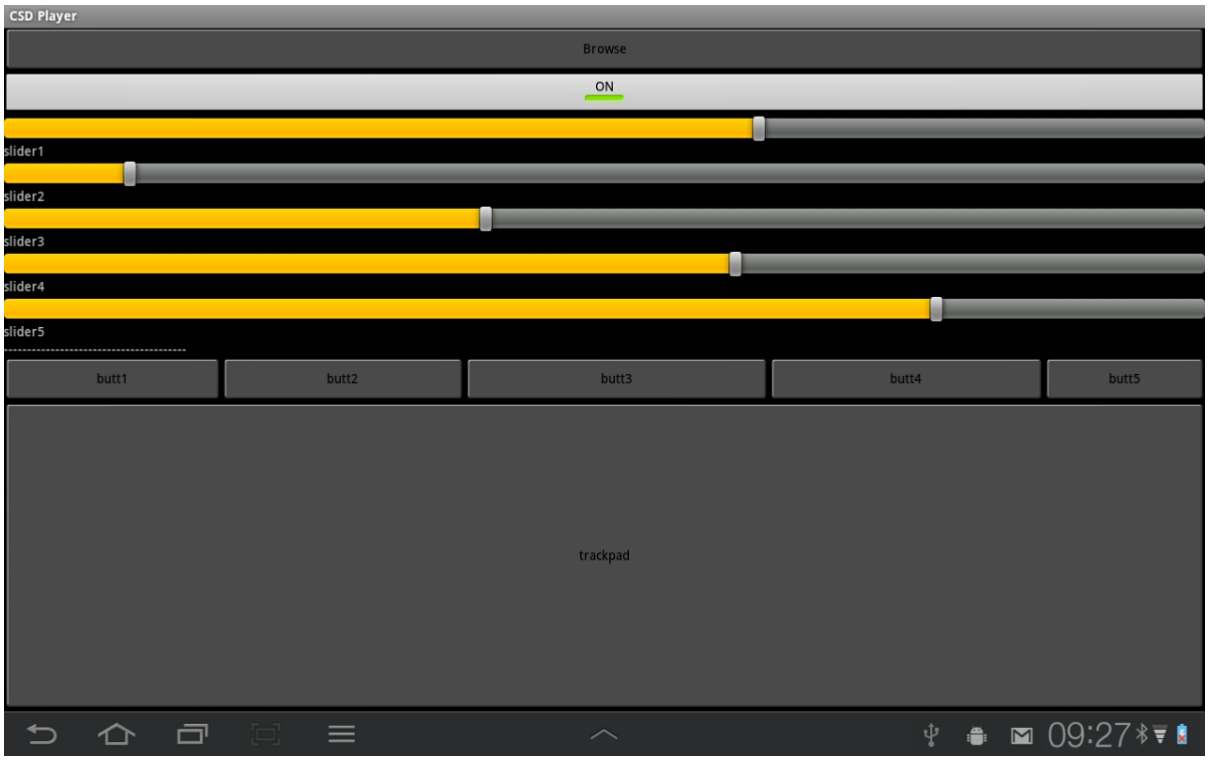

Figure 3: *CSDPlayer for Android, a general-purpose Csound performance application*

5.2. Other systems

Projects similar to MCP currently exist. There are two which are of note and comparable to Csound in terms of an extensive support for audio processing: libpd and Faust. The latter, already cited above, is notable in the sense that it provides a very defined algebra for the manipulation of block diagrams [16] and extensive meta-programming capabilities. It does not however attempt to implement an audio engine as such (i.e. it does not encapsulate the concept of a scheduler), which might be useful for some tasks. It instead relies on the target application to provide one. Its library of signal processing functions is also still underdeveloped. Although a new multirate processing version is being developed, currently it only supports time-domain single-rate processing.

The other project, libpd, is based on the Pd music programming system. Similarly to Csound 5, it offers Pd as a library with a C-language API, which can be wrapped in a number of languages. As with MCP, Android and iOS versions exist. Pd programs are generally edited graphically within the original Pd application, and stored in a text file, which, while human readable, is not very easily decoded. Such programs, or patches, contain the DSP code that is used by libpd; the library itself does not include the graphical element in Pd for patch editing. A significant difference with Csound is that at all steps of the way, Csound code is preserved as a completely human-readable, text-editable ASCII data. In addition, Csound offers a built-in score, so in addition to DSP instruments, whole compositions can be included in a mobile application. The Csound score is mostly a data format/specification, rather than a fully-develop programming language, but it has some simple support for value substitution and loops. Beyond these, there are other important differences between the languages themselves, but a discussion of these is beyond the scope of this paper.

5.3. Usefulness of Mobile Audio Applications

One final question for discussion regards the usefulness or lack thereof of audio applications on mobile systems. Over the years, it had become clear that the ideal environment for audio systems was the desktop computer (as a successor to the early UNIX-based graphic workstations). For the majority of tasks, especially the ones involving significant computational and digital audio hardware resources, it still is. However, mobile devices, especially tablet computers, have started to offer significant computing power, which is comparable to the previous generation of desktop systems. Digital audio encoders and decoders in some devices are of an acceptable quality (although not at the so-called professional grade) and external hardware is available in some cases, which can be of a higher specification.

This allows us to propose a number of scenarios that would be useful for digital audio research and development. There are already a number of commercial applications that take advantage of mobile platforms. Software synthesizers, effects processors, multitrack and loop-based sequencing programs, and interactive computer music instruments are some examples of the type of software that can be targeted at these systems. Innovative distributed computer music composition environments can be aimed at a cooperation between multiple computing platforms, which users might have at their disposal (phone, tablet, desktop computer).

For DAFx research, mobile platforms can be an excellent testbed for algorithms. Their portability might offer a flexible demonstration environment. Finally, the presence of ubiquitous audio com-

puting devices can potentially alter our approach to the design and application of algorithms for digital audio, in ways that are not completely predictable at this stage.

6. CONCLUSION

In this paper, we have introduced the MCP toolkit, which can provide the audio component for an agile development environment for mobile platforms. We have explored the individual attributes of the target platforms with regards to their audio infrastructure, Android and iOS. The software core of MCP, based on its CsoundObj class was discussed in detail with code examples. The iOS and Android implementations of MCP were presented and some application code samples were shown. This was followed by a consideration of the main applications of the technology. The paper concluded with a wide-ranging discussion of audio programming and languages for mobile applications. The source code for MCP and its sample applications can be obtained from the Csound 5 main GIT repository

git://csound.git.sourceforge.net/gitroot/csound/csound5

7. ACKNOWLEDGMENTS

We would like to acknowledge the support of An Foras Feasa via the Digital Arts and Humanities programme, which provided the funding for Steven Yi's contribution to this paper.

8. REFERENCES

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

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

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

[4] 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.

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

[6] V. Lazzarini, "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.

[7] J. Ousterhout, "Scripting: higher-level programming for the 21st century," *IEEE Computer*, vol. 31, no. 3, pp. 23–30, 1998.

[8] R. Damasevicius and V. Stuikys, "Separation of concerns in multi-language specifications," *Informatica*, vol. 13, no. 3, pp. 255–274, 2002.

[9] M. Mathews and J. E. Miller, *MUSIC IV Programmer's Manual*, Bell Telephone Labs, 1964.

[10] M Mathews, F. R. Moore, and J. C. Risset, "Computers and future music," *Science*, vol. 183, pp. 263–268, 1974.

[11] G. Loy, "The CARL system: Premises, history and fate," *Computer Music Journal*, vol. 17, no. 2, pp. 23–54, 1993.

[12] S. Pope, "Machine tongues xv: Three packages for software sound synthesis," *Computer Music Journal*, vol. 17, no. 2, pp. 23–54, 1993.

[13] Y. Orlarey, D. Fober, and S. Letz., "Faust: an efficient functional approach to dsp programming," in *New Computationals Paradigms for Computer Music*. 2009, Edition Delatour.

[14] M. Puckette, *The Theory and technique of computer music*, World Scientific Publ., New York, 2007.

[15] V. Lazzarini, J. Timoney, and T. Lysaght, "Spectral processing in csound 5," in *Proceedings of International Computer Music Conference*, New Orleans, USA, 2006, pp. 102–105.

[16] Y. Orlarey, D. Fober, and S. Letz, "An algebra for block diagram languages," in *Proceedings of International Computer Music Conference*, Berlin, Germany, 2002.