A Framework for Real-Time Digital Audio Processing in the Internet Browser

Will Strumolo
N187-495-26

Submitted in partial fulfillment of the requirements for the
Master of Music in Music Technology
in the Department of Music and Performing Arts Professions
in The Steinhardt School
New York University
Advisor: Juan Bello
[DATE:2008/6/18]
Abstract

In an attempt to address the lack of audio processing in web-based applications, various methods are explored for generating dynamic audio and real-time processing. After the logical platform is identified, a framework is theoretically and practically designed. The framework includes implementations that target the everyday web developer, as well as the digital signal processing programmer, in an effort to unite the two parties and infuse the next generation of rich Internet applications with audio processing technologies.
## Contents

1 Introduction .................................................. 5  
   1.1 Overview .................................................. 5  
   1.2 Motivations ................................................. 6  

2 History, Context and Related Work ............................ 7  
   2.1 Rich Internet Applications ................................. 7  
   2.2 Audio on the Web ........................................... 9  
   2.3 DSP Frameworks ............................................ 11  

3 Methodology .................................................... 14  
   3.1 Platform Assessment ........................................ 14  
   3.2 Flash Audio Architecture ................................. 17  
      3.2.1 Analyze native classes ............................... 17  
      3.2.2 Assess previous extensions ......................... 18  
      3.2.3 Account for future releases ......................... 19  
   3.3 Framework Design .......................................... 22  
      3.3.1 Distinguish black-box vs. white-box ................. 22  
      3.3.2 Integrate with native classes (black-box design) . 23  
      3.3.3 Abstract the development model (white-box design) . 26  

4 FLoundr: A Flash-based Framework for Digital Audio Processing ... 30  
   4.1 The Black-Box: Developing with FLoundr .................. 30  
      4.1.1 DynamicSoundChannel class .......................... 31  
      4.1.2 Processing ........................................... 33  
      4.1.3 Synthesis ............................................ 34  
   4.2 The White-Box: Developing for FLoundr ................... 35  
      4.2.1 Data interface ........................................ 35  
      4.2.2 Processing interface .................................. 36  
      4.2.3 Processing repository ............................... 37  
      4.2.4 Utilities ............................................ 38
4.2.5 Extending the \textit{DynamicSoundChannel} class \hfill 38

4.3 Example Applications \hfill 39
   4.3.1 White-box example: \textit{RingModulator} \hfill 40
   4.3.2 Black-box example: MP3Player \hfill 41
   4.3.3 Black-box example: SoundLab \hfill 42

4.4 Deployment \hfill 45

5 Conclusions \hfill 46
   5.1 Future Work \hfill 46

Bibliography \hfill 48
1 Introduction

1.1 Overview

We, as a technologically competent and dependent society, are by and large delighted with the new breed of software application that replicates desktop software functionality in the web browser—referred to as “floating” software applications. Suddenly, a host of everyday activities (word processing, contact management, presentation design, etc.) can be performed on any piece of technology with an Internet connection and a web browser. Even rich media applications, which have generally been considered too robust for web deployment, have seen permutations in this style; notably, video and audio playback have become two of the most popular uses of the web. And, as network technologies have steadily improved, we have begun to see media processing applications, like image and video sharing sites that allow for editing and other media manipulation. Distressingly absent from this new technology experience is any substantial use of audio processing, and that is the concern addressed in this thesis.

Platforms exist to handle audio in the Internet browser, and there is a large community of developers adept in these programming environments—so why is audio processing not generally implemented? Because there is no structured framework for doing so. The lack of a framework means that programmers are on their own when attempting to employ the necessary technologies for audio processing in the Internet browser. And, while there is an adequate knowledge-base on both building rich web experiences and constructing effective digital audio processes, the two communities do not easily connect, and any progress made on the individual level cannot be readily implemented by the community at large.
In this thesis, after analyzing the relevant technologies and theoretically defining the ideal framework, a solution will be proposed to the aforementioned problem, in a Flash-based DSP framework, called FLoundr, which provides seamless **black-box** integration for the rich Internet application developer, as well as a logical, proven **white-box** extension paradigm for the more low-level programmer (including those skilled in digital signal processing).

1.2 Motivation

Over the past decade we have been witness to the effects that powerful and affordable desktop software has had on the global community. A new generation of writers, artists, filmmakers, and musicians has risen in the wake and, through the advancement of network technologies, taken the world by storm. Recently, rich Internet applications have removed the need for the desktop entirely, and we stand on the edge of a new creative model, where collaboration and real-time productivity can flourish. Within this revolution, however, there is a concerning lack of audio processing tools, and I hope to address this issue by targeting the everyday web developer and offering a simple, elegant solution for audio handling in the Internet browser.
2 History, Context and Related Work

Dynamically processing audio in the web browser represents a convergence of three separate areas of software development. The first is the rich Internet application, which represents the final product’s browser-based packaging. The second is audio handling in the web browser, which presents the possible methods of rendering sound on the Internet. And the third is the digital signal processing (DSP) framework, which provides templates for designing effective audio processing environments. These three concepts are discussed below.

2.1 Rich Internet Applications

In the first decade or so of the World-Wide Web, the majority of applications built for the Internet browser were HTML-based. In this design, user interfaces (UI) were structured as a collection of static pages written in the HyperText Markup Language (HTML). Because of its simplistic design and its low-cost maintenance and deployment, this method of web development has remained the most popular, despite the archaic and unrewarding user experience it presents. Recently, however, network technologies have advanced to the point where other more robust and powerful web design strategies can be implemented without a significant increase in computational expense. The result of this advancement is a new breed of web application, known as the “Rich Internet Application” (RIA), which refines the user interface in order to provide a more dynamic web experience.

At the heart of the RIA is the user experience. The platforms put together to build these applications are concerned first and foremost with providing enough client-side intelligence to create a rewarding UI—something HTML is unable to do on its own. The result, after several
years of steady RIA development, is a collection of web UI technologies capable of replicating many of the most robust software experiences on the personal computer. This, in combination with the continued enhancement of server-side technologies and the steady increase in network connection speed, has made it possible to recreate almost any type of software that we enjoy on the desktop in the web browser.

This idea of a floating software application—one which lives on servers and is accessed through a web browser—is the focus of this thesis. Relevant examples include Google Docs1 (an office suite of word processing, spreadsheet and presentation programs), video editing sites (such as JumpCut2) and a myriad of music, photo and contact management tools (like those offered through Facebook3 and Myspace4). Of concern here is dynamic audio processing and its absence in these applications, despite the presence of other media-rich experiences on the web.

There are a mix of relevant technologies used to build RIAs, which include Microsoft’s Silverlight5, Adobe’s Flex6 and a collection of asynchronous JavaScript approaches (known generally as Ajax). Figure 1 shows where these technologies fall in the web application build path; their responsibility is to broker the relationship between server and client on their own terms and infuse the client itself with more intelligent behavior. These platforms will be revisited in section 3.1.

---

1 http://docs.google.com/
2 http://www.jumpcut.com/
3 http://www.facebook.com/
4 http://www.myspace.com/
5 http://silverlight.net/
6 http://www.adobe.com/products/flex/
2.2 Audio on the Web

Rendering audio—or any type of multimedia for that matter—has never been a concern for Internet browser architects. As network connection speeds have increased and multimedia (audio and video) has adopted the web as its primary host, a collection of third-party plug-ins has assumed this rendering responsibility. While market competition has resulted in a steady improvement of quality across these plug-ins, that same economic phenomenon has left any hope of universal browser-based audio handling behind; it seems we will have to move forward without a global standard and hope that a “higher audio-centric consciousness” among plug-in designers prevails [Iint02]. The following introduces a selection of the most relevant of these multimedia plug-ins and the technologies they employ.

The browser-based media plug-in has seen many incarnations. The idea of client-side audio synthesis was proposed early on by Ossenbruggen et al [Oss95], and has since been developed in many forms. Phil Burke designed JSyn\(^7\) for real-time sound synthesis in web-hosted Java applets; his method introduces a native Java API that calls disguised ‘C’ code to generate the audio [Burk98]. JSyn is perhaps the best example of an original API designed for browser deployment, but many other realizations exist. Another popular approach has been the

\(^7\) [http://www.softsynth.com/jsyn/](http://www.softsynth.com/jsyn/)
transformation of a proven desktop synthesis engine into a browser plug-in. Both Alonso et al [Al04] and Frauenberger et al [Frau03] have developed different methods in this manner, both using Miller Puckette’s Pure Data computer music system. The former method mounts the entire graphical programming language in the browser for real-time synthesis, while the latter masks the original environment to concentrate on processing Internet audio streams.

While these sorts of plug-in approaches are novel in design and present some of the more powerful browser-based audio handling capabilities, they lack widespread adoption and require developers to learn a unique set of tools. The popular solution to this problem of adaptability is a collection of proprietary plug-ins from leading software manufacturers, like Apple and Adobe. Apple’s QuickTime plug-in is used to deploy many different media formats. There is an accessible API, but many developers tend to settle for using QuickTime as little more than an embed tool. Adobe’s Flash plug-in is used to add interactivity and animation to web pages; its ability to handle streaming audio and video has made it a leader in media-rich application development. Other proprietary plug-ins are being developed (of note is Microsoft’s Silverlight), but QuickTime and Flash remain the leaders in the field. All of these popular plug-ins are guaranteed to be present in a majority of web browsers, and developing for each piggybacks on some of the more recognizable web programming platforms in use today, making them a logical host for a digital audio processing environment. These methods will be revisited in section 3.1 in an attempt to identify the most successful approach.

One additional method of handling audio on the Internet that does not fall into the previously mentioned categories is MPEG-4, which is an audio and video encoding standard. The

---

8 http://puredata.info/
9 http://www.apple.com/quicktime/
10 http://www.adobe.com/products/flashplayer/
collection of methods included in the very robust MPEG-4 format could potentially be accessed by a programming environment in order to create an audio processing web application, and compatibility across browsers would be guaranteed because the format is standardized. As of yet, however, low-level access to MPEG-4 is not regularly included in web application development environments, so a framework built on the format would be asking more of everyday web developers and thus is not ideal for the work presented here.

2.3 DSP Frameworks

To understand the concept of a DSP framework, its parts need to be defined:

(1) Digital signal processing (DSP) is a multi-disciplinary field focused on manipulating signal data and has many different applications. In this thesis, the concern is audio signal processing and how a DSP environment can be used to produce realtime digital audio effects.

(2) The term “framework” in this case is understood in the object-oriented software sense, as described by Ralph Johnson: “a framework is a set of classes that embodies an abstract design” [Joh91]. Frameworks can have practical functionality and include classes that are not abstract; what makes a framework a framework, however, is that it includes a theoretical model for how instances of these classes are meant to interact and forces adherence to this model through the presence of abstract classes.

A DSP framework as it is discussed here, therefore, is a strict design paradigm for implementing a signal processing environment, focused on audio and music (it should be noted that there are other non-musical DSP frameworks focused on audio signals, such as speech processing, but the concern here is musical audio). A successful, sustainable DSP framework
needs to be logical (to allow for effective and efficient development), flexible (to account for future adaptations) and stable (to ensure proper deployment).

There are several DSP frameworks that concentrate on audio and music; two of the most recent and most successful are presented here. Marsyas\textsuperscript{11}, originally written by George Tzanetakis, is an audio processing framework that is particularly adept at deploying Music Information Retrieval applications. Its architecture mimics a server-client relationship, where a C++ server contains the signal processing methods and Java-based client is responsible only for user interface functionality. The server structure is divided into two categories of classes: process-like (for transforming data) and data-structure-like (for holding data in different formats). Marsyas follows the traditional frameworks paradigm of including ample abstract classes to dictate build path and account for future expansion [Tza00].

CLAM\textsuperscript{12}, originally written by Xavier Amatriain, is a C++ framework with a similar design concept to that of Marsyas. It is based on Amatriain’s 4MS (Object-Oriented Metamodel for Multimedia Processing), which defines the architecture in theoretical terms. Like Marsyas, 4MS identifies two sets of processing objects: \textit{Processing} objects and \textit{Processing Data} objects. The former allows for the control and manipulation of data, while the latter holds data in static containers. Also like Marsyas, CLAM enforces model-adherence through the definition of abstract classes. In addition to the 4MS framework, CLAM also includes a collection of audio processing algorithms and packaging types. The combination of a solidified framework and a robust repository allow CLAM to behave in both a \textit{black-box} (implementing the repository) and a \textit{white-box} (extending the framework) manner [Ama07]. The dual nature of CLAM is a trait

\textsuperscript{11} http://marsyas.sness.net/
\textsuperscript{12} http://www.clam.iua.upf.edu/index.html
that needs to be successfully implemented in any framework hoping to attract both application
designers (who would use it as a *black-box*) and framework developers (who would use it as a
*white-box*). The concepts the *black-box* and the *white-box* are discussed in more detail in section
3.3.1.

There exist other audio and music processing frameworks similar to Marsyas and CLAM, but
these two represent the most modern, efficient and proven approaches at the time of this writing
and are thus the most relevant to the investigation.
3 Methodology

To adequately design the intended framework for browser-based DSP, the most effective platform had to be identified, understood and extended. This investigation required several distinct steps:

1. Platform Assessment
2. Flash Audio Architecture
   a. Analyze native classes
   b. Assess previous extensions
   c. Account for future releases
3. Framework Design
   a. Distinguish *black-box* vs. *white-box*
   b. Integrate with native classes (*black-box* design)
   c. Abstract the development model (*white-box* design)

3.1 Platform Assessment

In order to arrive at the most effective platform for processing audio in the web browser, the necessary criteria need to be laid out. If the primary goal is to build a self-sufficient framework that is easy to both implement (appeal to the RIA developers) and extend (appeal to the DSP developers), then the necessary criteria for the platform are: (1) a native sound engine so no additional plug-ins are required, (2) universal browser presence so that applications built on the framework can reach the most users, and (3) a widely-used programming environment so a large community of developers can be targeted.

Requiring a platform to have its own sound rendering system narrows the list of choices considerably. The majority of RIA approaches are occupied with bringing logic to the client in as efficient a manner as possible, ignoring the concept of multimedia entirely. That leaves us with the sound-capable plug-ins introduced in section 2.2: Adobe Flash, Apple QuickTime,
Microsoft Silverlight, Java-based implementations like JSyn and a host of lesser-known adaptations like those discussed using the Pure Data system.

Of these, the list can be further narrowed by examining how globally adopted these technologies are to see which will reach the largest audience. Adobe Systems, Inc. has commissioned annual studies for several years to track Plug-in technologies, the most recent of which was done by market research group Millward Brown\textsuperscript{13} in March 2008.

Details of the study’s methodology are publicly available \cite{Ado08} and the results are shown in Figure 2.

![Figure 2. Millward Brown Plug-in Survey, March 2008 \cite{Ado08}](image)

Apparent in these results is Adobe Flash Player’s popularity as a multimedia plug-in. The results of this survey are also supported by the platform’s presence as the driving technology in almost every video and music sharing site as the delivery platform of choice (a list that includes YouTube\textsuperscript{14}, LastFM\textsuperscript{15}, and others).

\textsuperscript{13} http://www.millwardbrown.com/
\textsuperscript{14} http://youtube.com/
\textsuperscript{15} http://www.last.fm/
Finally, of the technologies that satisfy the sound-rendering and universal browser requirements, only Flash possesses both its own scripting language (ActionScript) \textit{and} a dedicated vehicle for RIA deployment (Adobe’s Flex environment), both of which have a substantial developer community. Other technologies can be integrated with various design environments to produce similar results, but Flash is a completely self-sufficient platform, providing developers with all the necessary media, design, and logic functionality to build applications. This is why, of the technologies capable of handling sound in the browser, Flash is the only one which is also a popular RIA development tool.

All of this considered, Adobe Flash is the logical plug-in choice for building an audio processing framework for the web. The question remains, however, as to what platform should be used to access the plug-in. The primary methods of packaging Flash content are its scripting language ActionScript (AS) and the aforementioned Flex framework, which combines AS and a user-interface markup language, called MXML, to streamline RIA development. In addition there are several third-party implementations that access AS classes in a variety of ways. The common thread in all of these is their reliance on native AS classes. For this reason, the most effective framework will integrate directly with the native AS script, ensuring it can be used in the most Flash development environments possible. Furthermore, ActionScript’s current release, ActionScript 3.0 (AS3), is an almost fully object-oriented programming language, making it an effective platform for mimicking most relevant framework architectures. The choice of AS is also supported by evidence from the current web development community in the form of open-source libraries built on AS. Examples of these libraries include Tweener\textsuperscript{16} for controlling the

\textsuperscript{16} http://code.google.com/p/tweener/
movements of objects onscreen and Papervision3D\textsuperscript{17} for generating simulated 3D graphics. The success of libraries like these is a testament to ActionScript’s extendability and supports the idea of a Flash-based framework. The framework presented in this thesis, therefore, will be built in AS3 and will use the Flash Player browser plug-in for client-side audio handling.

3.2 Flash Audio Architecture

3.2.1 Analyze native classes

Implementing a framework in a proprietary environment like Flash can be difficult. Adobe encourages community development with freely available compilers and software development kits (SDK) with thorough documentation, but AS3 is not an open-source language—keeping the bulk of Flash Player’s build process a mystery. This lack of low-level knowledge on and access to the player will make any framework slave to the behavior of AS3’s native classes. For this reason, it is important to analyze the pertinent AS3 classes, in order to identify the most effective means of infiltrating the process. In this case, the classes typically involved in handling audio must be addressed.

There are three primary classes involved in bringing audio to an AS3 application. The first is the \textit{Sound} class, which loads audio data from a variety of sources and returns file information, like size and ID3 metadata. An instance of the \textit{Sound} class can be passed to an instance of the \textit{SoundChannel} class, which is used to play, stop and monitor a sound’s playback. And an instance of the \textit{SoundTransform} class can be applied to a \textit{SoundChannel} instance to alter volume and panning values. A diagram of this entire generalized process is in Figure 3.

\footnotesize
\begin{itemize}
  \item \textsuperscript{17} http://blog.papervision3d.org/
\end{itemize}

Strumolo 17
3.2.2 Assess previous extensions

For a long time there was little a developer could do to extend the audio handling capabilities of Flash Player. But with the release of Flash Player 9 in early 2006, an AS3 method called loadBytes was introduced. This method could be called on the Loader class (traditionally used for loading a Flash file within another Flash file). What the arrival of this method meant was that .SWF files (the format of Flash files) could be written in real-time byte by byte, and, if you could control content generation at the byte level and figure out the .SWF binary syntax for packaging sound, then you could generate dynamic audio on the fly.

A community of developers led by Andre Michelle [Mic07] did the necessary work, and soon they had perfected this sound “hack” and Flash synthesis was born. Several dynamic web audio sites have since incorporated this method, including SpliceMusic\(^{18}\) (a sequencer-based application) and the Hobnox AudioTool\(^{19}\) (a patch-based music synthesis environment). In addition developers Andre Michelle and Joa Ebert have made available an open-source library, called Popforge\(^{20}\), which allows interested parties to experiment with the hack. This method was

\(^{18}\) [http://www.splicemusic.com/](http://www.splicemusic.com/)
\(^{19}\) [http://www.hobnox.com/audiotool-startseite.1046.html](http://www.hobnox.com/audiotool-startseite.1046.html)
described in detail by Jordan Kolasinski [Kol07]; Figure 4 presents his block diagram of the Flash synthesis process.

Worth noting in this diagram is that none of the native AS3 sound classes are instantiated or called; its structure, while completely self-sufficient, is not designed to be implemented in the generalized build process for sound presented in Figure 3. Of course, all functionality is replicable at the byte level, and a framework using this hack would be possible, though a bit more taxing on the RIA developer, who would need to learn a new build process for sound.

3.2.3 Account for future releases

The original sound hack and the applications built on it have had substantial industry impact and have recently gotten the attention of the plug-in architects at Adobe. In May 2008 an early beta release of Flash Player 10 included an enhanced sound API for generating dynamic audio, meaning we will see more stable (not based on a hack) implementations of Flash sound in the near future.
The proposed API in Flash Player 10 adds two new functionalities to the *Sound* class. The first is an event titled *samplesCallback* that can be called on an empty instance of the *Sound* class; this event is dispatched at regular intervals asking for more audio data. Data is written to a *ByteArray* object which fills the empty *Sound* instance. This method can be used to synthesize sound by filling arrays with waveforms and reading them at a controlled speed. The second new functionality is a method titled *extract()* that can be called on a *Sound* object [Uro08_2]. The method returns a *ByteArray* of specified length and starting position that represents the audio data in the targeted *Sound* object. The returned data can then be written to an empty *Sound* instance using the aforementioned *samplesCallback* event and we have native dynamic audio generation in Flash [Uro08_3]. What these two functionalities allow is the stable deconstruction and reconstruction of a loaded audio file and a defined access point (between deconstruction and reconstruction) for processing the resulting data. This new sound API is represented in pseudocode in Figure 5.

```plaintext
var sound1 is the sound to be extracted
var sound2 is an empty sound to be written to
var buffer is the buffer size

function samplesCallback
    extract buffer samples from sound1
    fill an array with extracted samples
    for length buffer
        write to sound2 from the array
    end loop
end function

call samplesCallback function on sound2
```

*Figure 5. Flash Player 10 Dynamic Sound Generating Pseudocode*

The API is presented in greater detail in the following section during the discussion on framework design (section 3.3).
It should be noted that the beta release of Flash Player 10 is far from final, and the current realization of the sound API has its shortcomings. For instance, the refresh buffer in the `samplesCallback` event is constrained to values between $2^9$ (512) and $2^{13}$ (8192), where shorter buffers tax the application more and longer buffers produce greater latency. This is a significant but addressable problem. More troubling, however, is an overall latency issue caused by an internal buffer of 0.2 to 0.5 seconds that is put in place to prevent application dropouts. This large-sized latency will prevent some timing-specific uses of dynamic audio (where exact timing is critical), but it is an acknowledged source of concern by Adobe engineers and will hopefully be addressed in future Flash Player releases. And, finally, the new API can only accept audio that is a stereo MP3 file, sampled at 44.1 kHz [Uro08_2]. Obviously, a strict file format (especially a compressed one) limits the processing capabilities. But these sorts of restrictions are often put in place to test the functionality of a new API. The ability of Flash player to generate different file types at different sample rates has already been proven by developers using the sound hack presented in the previous section; so it seems logical that the new AS3 Sound API will incorporate these in future releases.

Even in its infancy, the enhanced sound API included in Flash Player 10 represents the most stable and predictable method of accessing raw audio data in the platform. Combined with its logical implementation into the AS3 build process for sound, it makes for the safest and most easily utilized API for dynamically processing audio in Flash. For these reasons, the framework presented here will be built on Flash Player 10.
3.3 Framework Design

At this point adequate research has shown the most effective platform for an audio processing framework to be ActionScript 3.0 and the Flash media plug-in. And an analysis of the current development environment has shown that the most logical framework will integrate with the AS3 build process for sound by utilizing the enhanced sound API in Flash Player 10. What is left is to design and build the framework itself on these established terms. Following the model of CLAM presented in section 2.3, the resulting framework will exist as both a black-box designer utility, providing smooth integration with the traditional build path, and a white-box extendable environment, with forced compliance to an abstract framework architecture, as described in section 2.3, to ensure successful and lasting expansion. Definitions of these behaviors and approaches for implementing them are described in detail below.

3.3.1 Distinguish black-box vs. white-box

In the framework model presented here, the final product will exist two ways: as a black-box and as a white-box. In the black-box implementation, all the inner-workings of the framework are hidden. In this case, the user is given a simple method for integrating the framework into his own applications and is never concerned with what is happening inside. In the white-box implementation, every aspect of the framework is exposed. In this case, the user can access any aspect of the framework he chooses and construct his own method of integration. Accounting for both implementations is crucial to the framework presented here because the goal is to connect two different user bases. The first is the community of RIA developers who will build dynamic audio applications for the web browser; these users are concerned only the black-box...
implementation and how the framework fits into their applications. The second user base is the community of DSP programmers who will write algorithms for processing audio in the framework; these users access the framework as a *white-box*, interacting with low-level elements and updating it with their own processes. Both groups of users are needed in order for the framework to grow and have a lasting effect, and so well-defined framework implementations of both a *black-box* and a *white-box* are needed.

### 3.3.2 Integrate with native classes (*black-box* design)

There are several concerns in identifying the optimal point where the framework should penetrate the generalized build process for sound in AS3 and act as a dynamic audio *black-box* for Flash-based RIA developers. First, the target area needs to allow for seamless integration at run-time; that is, any processing applied to the audio cannot interrupt the normal rendering process. This sounds straightforward, but accounting for it in the Flash Player 10 sound API reveals significant target area limitations. Specifically, the various latency issues discussed in section 3.2.3 show that accurate syncing in the audio generation process is crucial. To ensure that audio is extracted, processed and written in appropriate fashion, these three actions need to be performed within the same event buffer. As a practical test, the `extract()` method was moved to a separate but identical event buffer and both were called on the same `Sound` class and deployed simultaneously. The concept is presented in pseudocode in Figure 6.
function event1
    extract sound information
end function

function samplesCallback2
    write sound
end function

call event1 and event2

Figure 6. A Separated Audio Generation Test Pseudocode

The result of this two-buffer method was a collection of run-time errors, thus proving the need to enclose all audio handling in a single event buffer in order to ensure run-time success.

The previous investigation has helped define the point of penetration as a single event buffer, but placement of calling that event has yet to be determined. The primary goal in a black-box system is logical integration; the user should put something logical in and get something logical out. Considering the build process for sound presented in Figure 3 and not wanting to remove any of the functionality of existing native classes, the most effective point of penetration is after a Sound class has been instantiated but before that instance is passed to a SoundChannel object. This placement will allow users to maintain use of the Sound and SoundChannel classes. An updated build process showing the framework penetration location is in Figure 7.

Figure 7. Framework Penetration Location in AS3 Sound Structure
While Figure 7 represents the dynamic sound build process in theory, practical implementation requires slight adjustments to the behavior. Specifically, the SoundChannel class can only be passed an instance of the Sound class, so it will not accept the dynamic audio generated by the framework. A simple subclass of the SoundChannel class could effectively resolve this issue, extending its behavior to include the acceptance of dynamic audio, but the proprietary and commercial nature of the plug-in has resulted in almost all of the native AS3 classes being declared final, including all those that are sound-related. This effectively prevents any traditional inheritance solutions for the problem. In object-oriented programming, the alternate choice to an inheritance structure is the idea of composition. Composition replicates the concept of hierarchy by instantiating parent classes within child classes (as opposed to extending parent classes within child classes). In this case, we need to define a new class which instantiates the SoundChannel class and defines it publicly so that SoundChannel methods may be called. With this final adjustment we have solidified the theoretical placement of the framework within the AS3 sound structure and integrated it with the appropriate native classes. Figure 8 shows the resulting black-box behavior.

Figure 8. Framework’s Black-Box Design
One additional clarification is necessary regarding the infiltration point, and that is concerning the AS3 \textit{SoundTransform} class—the name of which might be misleading when implementing audio processing. This class is traditionally used in the generalized build process for sound to adjust volume and panning information and, therefore, could theoretically be used for sample-by-sample audio effects. Adobe engineers, however, opted not to give developers access to the \textit{SoundTransform} class when updating the sound API for Flash Player 10 and to instead focus the implementation of dynamic audio around the \textit{Sound} class. This method not only allows audio to be synthesized but also provides an infiltration point that can handle block processing, a necessary element of any successful DSP platform.

### 3.3.3 Abstract the development model (white-box design)

There are several aspects of a framework that will allow it to be accessed and extended in the \textit{white-box} model. The first is the logical design of its architecture. Both DSP frameworks presented in section 2.3 (CLAM and Marsyas) employ a two-“class-type” structure. In these frameworks, classes are either of a \textit{transformation} type or a \textit{data-holding} type. Transformation classes in a signal processing framework are responsible for all manipulation processes applied to signal data. Data-holding classes are passive in nature and are responsible for holding data in a designated format. Combined, these two class types can accomplish almost all of the signal processing required by any platform. This generalized, dual-category DSP architecture model is a proven, widely-adopted model, and thus a new DSP framework stands the greatest chance of development popularity and longevity by satisfying this basic paradigm.
Both CLAM and Marsyas expand on this model, but the lack of flexibility in the proprietary AS3 platform makes a Flash-based framework require much less definition. For instance, because all communication protocols are natively defined and strictly deployed, a designated class-type for controls is not necessary. The same can be said for a class-type dealing with timing events—this sort of low-level access is not readily accessible and needs no classification in the framework. Therefore, the framework presented here will use two class-types: the \textit{Data-Holding} classes, which will define various formats for the extracted audio as it moves through the framework, and the \textit{Transformation} classes, which will encompass any functionality that manipulates the data in the \textit{Data-Holding} classes.

For a framework to be a successful \textit{white-box}, its interior class structure needs to be logically integrated into the generalized \textit{black-box} presence in the native environment. In other words, developers wishing to treat the framework as a \textit{white-box} will find that the insides connect logically and seamlessly to the outsides. In this framework, the interior classes must exist between the \textit{extract()} method and writing audio data to the output; in between these two activities, data can be packaged and held in \textit{Data-Holding} classes and manipulated using \textit{Transformation} classes before being output to the speakers. A \textit{white-box} diagram of the framework is shown in Figure 9.
Finally, to ensure logical, flexible and stable use, a framework must force adherence to its interior structure by defining a strict class-type scheme through the use of abstract classes. In object-oriented programming abstract classes are incomplete classes which instantiate nothing but contain methods and properties. A child of an abstract class is required to override these methods and properties—this concept is used in a framework to make sure future classes include the necessary elements. Both CLAM and Marsyas were designed on a set of strict abstract classes.

Unfortunately, while ActionScript 3.0 is an object-oriented language, it does not currently allow the use of abstract classes [Mo07]. The solution is to replicate abstract class behavior by using *interfaces*. Interfaces can mimic most of the functionality of abstract classes but they lack the force. Abstract classes *require* a set of behaviors, while interfaces simply *recommend*. This issue would be magnified if the framework defined many abstract classes; in this case, however,
only two class-type definitions are needed (*Data-Holding* and *Transformation*) and it is not asking the developer a lot to remember to implement the respective interfaces. Nevertheless, an interface-based architecture is not as stable as an abstract-based one, so an effort will be made to update the framework if and when abstract class support becomes available in AS3. That said, one advantage to using interfaces is the fact that they can be unambiguously instantiated. For example, if the framework needed to move a collection of *Transformation* classes whose specific definition would not be known at compile-time, it could initially send a collection of *Transformation* instances, and the classes implementing that interface could replace those instances later—an object-oriented concept known as *polymorphism*.

At this point the theoretical design is complete. The following section presents the developed, fully-functional framework and its practical implementation.
4 FLoundr: A Flash-based Framework for Digital Audio Processing

This section presents a Flash-based framework for digital audio processing in the web browser, called FLoundr (an amalgamation of FLash sOUND processeR). FLoundr is written in ActionScript 3.0 and has two primary uses: In its simplest incarnation, FLoundr can be used as a black-box by web (RIA) developers to transform their sounds into dynamic audio streams and provide them with a vehicle for applying digital audio effects. In addition, FLoundr also employs a proven framework for open-source, community-based enhancement by DSP programmers, a white-box implementation. As a framework, it is meant to be as simple to develop with as it is to develop for. FLoundr also includes within itself a small library of processing algorithms (the FLoundr processing repository), and it is hoped that with community development this collection will grow into one of the framework’s greatest assets. The details of FLoundr’s architecture are described below.

4.1 The Black-Box: Developing with the Framework

FLoundr’s design is primarily concerned with ease of integration. The classes are programmed exclusively in AS3 and can be imported into the build path along with the rest of the Flash library. Areas of the framework that concern the black-box user are presented below.
4.1.1 DynamicSoundChannel class

At the heart of FLoundr is a single parent class, called DynamicSoundChannel, which is responsible for deploying the entire dynamic audio process. As it performs a majority of the framework’s functionality, the class is introduced here in detail and can be accessed in full at the FLoundr website.

**Constructor Parameters:** DynamicSoundChannel has four possible constructor parameters. The only required parameter is a signal of some sort—this could be an instance of the Sound class, a url address of an MP3 (through AS3’s URLRequest class), or an instance of FLoundr’s SynthObject class (see section 4.1.3). This signal is the sound that will be translated into a dynamic stream. The second constructor parameter is a digital audio process of some sort; users can pass any class that implements the Processing interface (see sections 4.1.2 and 4.2.2), including any effect found in the FLoundr Processing repository. This is how users can pair an audio effect with the sound they wish to process; send the two as constructor arguments to a new instance of DynamicSoundChannel and you have an effect being applied to your sound. To allow more flexibility, users have the option of passing a collection of several audio effects instead of a single one. In this case, the user fills a vector with instances of audio effects and passes it the DynamicSoundChannel; this vector is then read and the effects are applied in sequential order. Not only does this allow users to apply more than one effect to the same sound, but it also allows them to reorder, add and remove effects in real-time by manipulating the vector and updating the DynamicSoundChannel’s processing instance variable. The default value for this second constructor is a blank process that does not affect the sound at all. The remaining
two constructor parameters allow users to designate the buffer size (between 512 and 8192 samples) and specify whether they want the sound to loop or not.

**Instance Variables:** There are two important instance variables in the *DynamicSoundChannel* class. The first has already been introduced: the *processing* instance variable. Users can update this variable at any time to dynamically control which processes are applied to the sound. The second is the *channel* instance variable. The *DynamicSoundChannel* automatically creates an instance of the AS3 *SoundChannel* class and wraps the audio within it. This use of composition allows a user to access all the instance methods of that class, which includes the *transform* instance variable for volume and panning manipulation.

**Instance Methods:** The only public methods in the *DynamicSoundChannel* class are *play()* and *stop()* . The rest of the functionality is accessed through the *channel* instance variable.

The *DynamicSoundChannel* class is, in essence, the embodiment of the *black-box*. Figure 10 updates previous block diagrams to show FLoundr’s *black-box* design and how it is incorporated into the build process for sound in AS3.

![Figure 10. FLoundr Integration Block Diagram](image-url)

Strumolo 32
To deploy FLoundr in their applications, users need only create an instance of it and pass it a signal in order to generate dynamic audio. Figure 11 shows the code needed for a basic integration of FLoundr.

```javascript
var sound:Sound = new Sound();
//FLoundr code follows
var process:Processing = new Processing();
var dsc:DynamicSoundChannel = new DynamicSoundChannel(sound, process, 1024);
dsc.play();
```

*Figure 11. Basic FLoundr Integration Code*

4.1.2 Processing

FLoundr’s audio processing classes all employ the *Processing* interface. This structure is for framework developers (*white-box* use) and is discussed in more detail in section 4.2.2.

FLoundr’s *black-box* users never directly work with the *Processing* interface; rather, they use various classes in the FLoundr processing repository that implement the interface. Using these classes requires little more than creating an instance and providing the necessary constructor arguments. Processing parameters are controlled through a collection of instance variables that can be updated in real-time. The constructor parameters and the instance variables available will change from effect to effect, so it is the duty of the user to read the comments in the class in order to successfully administer the process. Figure 12 gives generalized code for updating a process.
4.1.3 Synthesis

The third and final class to be used in the FLoundr black-box implementation is the SynthObject. SynthObject is a parent class that provides proper packaging of waveforms so they can be played in an instance of the DynamicSoundClass. Again, users would not directly call SynthObject but would instead instantiate one of its children, such as SineWave, SquareWave, etc., which are included in the framework. The only required parameters for these classes are frequency and amplitude values. Because dynamic audio in Flash currently only supports generation at a 44.1 kHz sampling rate, the global sampling rate variable is inaccessible at the moment. In the future, this sample rate would be included in SynthObject’s constructor parameters. The SynthObject was originally built to utilize various oscillators for the digital audio processes in the FLoundr processing repository. FLoundr is not meant to be a full-service synthesis environment, but these classes provide the basics. Code for playing a continuous sine wave is in Figure 13.

```java
var sound:Sound = new Sound();
//FLoundr code follows
var process:SomeProcess = new SomeProcess();
process.someParameter = 0.5;
var dsc:DynamicSoundChannel = new DynamicSoundChannel(sound, process);
dsc.play();

process.someParameter = 1.0; //update process parameter
dsc.processing = process; //apply updated process
```

Figure 12. FLoundr Update Processing Code

```java
//FLoundr code follows
var sinewave:SineWave = new SineWave(440, 0.5);
var dsc:DynamicSoundChannel = new DynamicSoundChannel(sinewave);
dsc.play();
```

Figure 13. FLoundr Continuous 440Hz Sine Wave Code
This concludes the very basic black-box implementation of FLoundr. As a black-box, it is
designed to work seamlessly with the traditional sound API in Flash and give users the simplest
and most effective controls for processing the audio. Of course, being open-source, FLoundr
allows low-level access and encourages framework development in the white-box model. A
description of that implementation of FLoundr will be presented in the following section.

4.2 The White-Box: Developing for the Framework

In addition to ease of integration, FLoundr is also concerned with ease of expansion. One of the
goals of the framework is to attract audio algorithm programmers, and, to do that, a logical class
architecture needs to be in place. Mimicking several established DSP frameworks that
concentrate on audio and music as discussed in section 3.3.3, FLoundr contains two primary
types of classes: those that implement the Data interface and those that implement the
Processing interface. These are addressed in more detail below.

4.2.1 Data interface

The Data interface is the realization of the Data-Holding class-type introduced in section 3.3.3
and describes the packaging of data for manipulation in the FLoundr framework. In this initial
realization of FLoundr, there are two classes that implement the Data interface.

The native AS3 method for extracting sound (Sound.extract()) returns an array of bytes;
while bytes are the most efficient data type (they do not require transformation at compile time
because they are already packaged in the necessary binary format), they are not conducive to the
majority of digital signal processing algorithms. For this reason, a loop to transfer this array into
float values is included in the *DynamicSoundChannel* class. The resulting float values could suffice as the base data type (most DSP math will need to access float values anyway) but they do not account for stereo channel packaging. Indeed the way float values are extracted from the file is one by one, left followed by right. Rather than asking developers to account for this on their own terms (which could result in entire systems based on moving at increments of two), it makes sense to package simultaneous left and right float values into one data container. In FLoundr, this is accomplished using the *StereoSample* class.

The other method for packaging data accounts for the buffer-based behavior of the dynamic sound API. Because the `extract()` method returns an array, FLoundr will receive a collection of values—not one at a time. So a *SamplesPackage* class exists, implementing the *Data* interface and holding a vector of *StereoSample* instances. The *SamplesPackage* class is the data container that is sent to and manipulated by *Processing* classes.

The *Data* interface will have more use in the future as more complex signal processing algorithms require modified data packages.

### 4.2.2 Processing interface

The *Processing* interface is implemented by every audio effect class in FLoundr and is how the framework infiltrates the dynamic audio generation process in Flash. The necessity of generating dynamic audio entirely within a single event buffer (as discussed in section 3.3.2) makes it impossible to separate the build process. This means that all data processing needs to be applied within that same event buffer. In order to successfully design the *DynamicSoundChannel* class to accept all sorts of processing classes during that event,
polymorphism is needed. The result is that the Processing interface is used as the type declared within the event buffer and any class implementing it can be loaded into the DynamicSoundChannel class.

The Processing interface itself defines only one method that must be overwritten for successful implementation. This method is called load() and requires a SamplesPackage as its parameter. This load() method is what is called on the undefined Processing type within the DynamicSoundChannel class. Additionally, the method returns nothing; the SamplesPackage remains intact—simply altered by the process.

Processing class developers need to understand, at the very least, the data flow within FLoundr. The black-box user (the programmer who instantiates DynamicSoundChannel and pairs it with an audio effect) will choose a refresh buffer time between 512 and 8192 samples. This buffer time will determine the size of the SamplesPackage that is passed into the effect. Classes therefore should account for receiving data in chunks—a fact that will make block-processing algorithms much easier to implement for DSP programmers.

The Processing interface is a crucial element of both FLoundr’s integration, and its expansion. Because of this, it will likely see the most attention, criticism and revision upon the framework’s release.

4.2.4 Processing repository

Language specific signal processing algorithms are an important part of application development. Programmers uncomfortable porting code between languages will often abandon their goal if there is not a language specific paradigm readily available, especially if the subject
of the code (in this case DSP) is one they are unfamiliar with. This is why the FLoundr processing repository may be one of the framework’s greater assets. ActionScript developers often come from design backgrounds (the platform, itself, evolved from a design-based software) and are some of the quickest to exhibit the aforementioned behavior. For this reason, a collection of DSP code in AS3 could have an affect on the community outside of the framework itself. With this in mind, the FLoundr processing repository will be treated as its own entity when the framework is released.

4.2.3 Utilities

A group of utilities, such as the oscillators built using SynthObject, may at one point require their own interface. At this time, however, the effects written using the Processing interface do not demonstrate the need for an abstract utilities class-type. As algorithms are added and new developers arrive, trends in programming may show a need for a standardized subclass for utility-like processes.

4.2.3 Extending the DynamicSoundChannel class

The structure of the DynamicSoundChannel is presented in section 4.1.1, but it should be noted that the class is not declared final and can be extended by developers. The source code included in the release of FLoundr is thoroughly commented and needs not be presented here in detail. The one area worth highlighting is the event buffer (represented by readSound()), which is shown in Figure 14.
This function needs to remain intact in order to ensure proper audio generation at run-time. The three processes included here are (1) extracting audio data (`extractSound()` function), (2) processing the data (`applyProcessing()` function), and (3) writing the data to sound (`playSound()` function). Developers wishing to restructure the `DynamicSoundChannel` class should be aware of the importance of this design and know that all three processes must be called within the same event.

This concludes the basics elements of FLoundr’s *white-box* implementation. It is a straightforward, bare-bones framework meant to be as flexible as possible without hurting its ability to expand. Thorough documentation and a shared web portal will accompany FLoundr’s release and attempt to foster community-driven development.

### 4.3 Example Applications

In an attempt to evaluate the practical implementation of FLoundr, two example applications were built. In the process, a small collection of processes accumulated in the FLoundr processing repository. The example applications and one of the processing classes are presented here.
4.3.1 *White-box example: RingModulator*

In order to create test applications using the FLoundr framework, some audio effect classes were needed in the FLoundr processing repository to demonstrate dynamic sound processing. The resulting collection includes envelopes of various kinds, several equalization processes, delay-based effects like a flanger, and a more musical effect in a ring modulator. The high speed at which these classes were designed was due in large part to presence of similar algorithms in other open-source frameworks, notably the AS3-based Popforge sandbox. Whenever possible, classes from these and similar libraries were adapted for FLoundr. The transfer process went surprisingly smoothly and demonstrates the ease of porting desired code and wrapping it for FLoundr. To demonstrate the build process of an effect class (a *white-box* implementation of FLoundr), the design of a simple ring modulator is described here.

Ring modulation is a straightforward signal process, in which two signals (one of which is usually a simple waveform like a sine wave and is classified as the “carrier” signal) are multiplied together [Zol02]. While unexciting at best, ring modulation does result in unmistakable audio processing and accomplishes this in the simplest of processes—perfect for a demonstration of FLoundr’s *white-box* implementation.

To build a ring modulation class in FLoundr, all that is needed is a generated sine wave to multiply with the input *SamplesPackage*. Conveniently enough, there is a *SineWave* class of the *SynthObject* type already included in FLoundr. To provide some user control, the sine wave frequency is made public. Conceptually, the class works like this: (1) each time it is instantiated a frequency value is calculated from user input and a sine wave instance is packaged in a vector; (2) then each time the *load()* method is called a loop runs across the input *SamplesPackage*
multiplying every *StereoSample* (both left and right) with the corresponding sine wave vector value. The only complexity in this program is handling the timing of the sine wave, but that issue can be resolved by implementing its own dedicated counter. The resulting effect class renders successfully. It can be heard in the application presented in section 4.3.3, and its source code accessed on FLoundr’s website.

4.3.2 *Black-box example: MP3 Player with EQ*

The ultimate test ground for any browser-based audio platform is the MP3 player. Flash-based MP3 players are by far the browser standard, present on every music site, artist page and personal weblog. Thus a MP3 player is an appropriate first application to test FLoundr’s *black-box* implementation.

In determining a build model for the player, care was taken to see how much of the application could be out-sourced—combining work already done in MP3 player design (by RIA developers) with existing code for audio processing effects (by DSP developers) would demonstrate the simplest way of using FLoundr. To this effect, a basic MP3 player design was replicated (a short playlist with song information, along with play and stop buttons) and enhanced to include audio effect sliders. Simultaneously, an audio effect written by a web developer already working with Flash Player 10 was located; the developer was sharing a 3-band EQ class [bli08], and this was reworked to adhere to the FLoundr framework. Finally, a simple MP3 logic class was written to combine the UI with the audio effect; the resulting class loaded audio into FLoundr’s *DynamicSoundChannel* class and processed it with the adapted 3-band EQ effect (now present in the FLoundr processing repository), while listening for user input through
the buttons and sliders in the UI. The result: a traditional Flash-based MP3 player with dynamic audio control in the form of a 3-band EQ. The player represents the most basic use of FLoundr as a black-box and demonstrates the ease of integration by using as many pre-built elements as possible (in this case, the UI and the audio effect). The player is included with the FLoundr release and can also be found on the FLoundr website. Figure 15 shows a screenshot of the final application.

**Figure 15. FLoundr MP3 Player with EQ Screenshot**

### 4.3.3 Black-box example: FLoundr SoundLab

To really demonstrate the power of FLoundr as a black-box and demonstrate a more advanced use of the framework, a graphical patch environment was built to allow user control over multiple audio effects—the resulting application being called the FLoundr SoundLab. The details of designing the interface are not particularly relevant, but a quick description on the end behavior will help. The interface is composed of six boxes—a sound box for generating sound (on the left), a speaker box for playing sound (on the right) and four different effect boxes for applying effects (in the boxes at the top). These boxes include patch chords which can be
connected in any order. When the sound box is connected (either through effects or not) to the speaker box, sound will play. Double-clicking on each box reveals a set of controls. The sound box offers six different sound sources to choose from. The speaker box displays stereo levels and offers volume and pan controls. And each of the four effect boxes include controls for their various parameters. The four included effects are: a stereo delay (with mix, feedback and gain controls for both channels), a flanger (with controls for delay, depth, speed, feedback and mix), a parametric EQ filter (with a graph-based control for simulating frequency, gain and Q values), and the ring modulator presented in section 4.3.1 (with carrier frequency control).

The SoundLab application demonstrates several aspects of FLoundr. First and foremost, it demonstrates audio processing in the web browser, which was the initial criteria for the framework. Second, it provides a visual representation of how FLoundr integrates with the traditional Flash process for rendering sound: The sound box uses AS3’s native `Sound` class to package the selected audio; similarly, the speaker box applies AS3’s `SoundTransform` class to the instantiated `SoundChannel` within FLoundr to affect volume and panning. The behavior of both these boxes is true to their AS3 classes—showing how FLoundr implements with the established process, instead of altering it. Finally, SoundLab also demonstrates the use of multiple `Processing` instances on one `DynamicSoundChannel`. At compile time a vector of four blank processes (FLoundr has a `BlankProcess` for this occasion) is loaded into the `DynamicSoundChannel`. When the application is run, the processes in the vector are applied in ascending order, and it is possible to replace any or all of these `BlankProcess` effects with one of the real audio effects included (stereo delay, flanger, parametric EQ or ring modulation), by accessing the `DynamicSoundChannel.processing` instance variable. Thus you can add effects
when they are connected on the screen, remove them when they are disconnected and reorder
them at will. Simple example code for adding an effect (the ring modulator) as the first effect to
be applied in the SoundLab is presented in Figure 16.

```java
    dsc.processing[0] = ringModulator;
```

*Figure 16. Changing an Effect in FLoundr SoundLab*

The user interface for the SoundLab was initially designed to demonstrate a stable *black-box*
use FLoundr. A future version, however, could be restructured for *white-box* integration,
allowing developers to test their FLoundr processing classes in a graphical environment. The
current version of SoundLab is included with the FLoundr release and can be found on the
FLoundr website. Figures 17 and 18 show screenshots of the application.

*Figures 17. FLoundr SoundLab Screenshot*
4.4 Deployment

In order to facilitate streamlined, efficient community development, FLoundr is hosted on its own designated website\(^\text{21}\). This portal includes documentation, examples and a packaged version of the framework in its most recent form—meant primarily for black-box users. At the same time, the project source code is hosted in a proven environment for open-source collaboration and accessed through subversion (a revision control system for software development)\(^\text{22}\)—meant primarily for white-box users. Care will be taken to update the packaged version on the FLoundr website in an appropriate fashion. The FLoundr website also acts as the appendix for this thesis, including all code discussed.

---

\(^{21}\) [http://www.floundr.org/](http://www.floundr.org/)

\(^{22}\) [http://code.google.com/p/floundr/](http://code.google.com/p/floundr/)
5 Conclusions

An investigation into dynamic audio processing in the web browser produced the following:

- Identified an open-source, extendable DSP framework targeted at the everyday web developer as a possible solution to the lack of audio processing in rich Internet applications.
- Assessed relevant browser plug-ins and development platforms and showed Adobe’s Flash Player and its ActionScript programming language to be the most widely adopted and the most adept at handling audio.
- Analyzed similar audio-based DSP frameworks to find the most proven architecture, identifying key traits to be (1) a two type-class hierarchy with data classes for passive containment and transformation classes for manipulating data and (2) the coexistence of both white-box and black-box framework implementation.
- Presented FLoundr: a flash-based DSP framework that integrates seamlessly with AS3, mimics key concepts of DSP frameworks, and brings real-time audio signal processing to the web browser.

5.1 Future Work

The FLoundr framework for web-based DSP is still in its infancy, so the primary objective at the time of this writing is to facilitate the growing community. This community will introduce important modifications and dictate much of the future work.

In terms of FLoundr’s black-box maintenance, steps will periodically need to be taken to ensure its compliance with the proprietary Flash plug-in. This process will be especially important during the early stages of FLoundr’s initial release because it is built on the beta version of Flash Player 10, utilizing and brand-new API (both of which are subject to frequent updates). Other black-box improvements could include additional examples, video tutorials, getting started documents and anything that might make the framework more appealing and easier to implement.
In terms of FLoundr’s *white-box* maintenance, much of this will be community-handled, but, while the community is being built, there is an immediate need for several classic DSP building blocks, such as convolution and FFT functions, whose presence will help attract additional developers. These classes will be one of the first additions to the FLoundr framework.
Bibliography


