An Audio Processing Library for Game Development in Flash

Raymond Migneco¹, Travis M. Doll¹, Jeffrey J. Scott¹, Christian Hahn², Paul J. Diefenbach², and Youngmoo E. Kim¹
¹Music and Entertainment Technology Lab¹; RePlay Lab²
¹Electrical and Computer Engineering; ²Digital Media Program
Drexel University, Philadelphia, PA, USA
Email: {rmigneco, tdoll, jscott, cmh66, pjdief, ykim}@drexel.edu

Abstract—In recent years, there has been a sharp rise in the number of games on web-based platforms, which are ideal for rapid game development and easy deployment. In a parallel but unrelated trend, music-centric video games that incorporate well-known popular music directly into the gameplay (e.g., Guitar Hero and Rock Band) have attained widespread popularity on console platforms. The limitations of such web-based platforms as Adobe Flash, however, have made it difficult for developers to utilize complex sound and music interaction within web games. Furthermore, the real-time audio processing and synchronization required in music-centric games demands significant computational power and specialized audio algorithms, which have been difficult or impossible to implement using Flash scripting. Taking advantage of features recently added to the platform, including dynamic audio control and C-compilation for near-native performance, we have developed the Audio processing Library for Flash (ALF), providing developers with a library of common audio processing routines and affording web games with a degree of sound interaction previously available only on console or native PC platforms. We also present several audio-intensive games that incorporate ALF to demonstrate its utility. One example performs real-time analysis of songs in a user’s music library to drive the gameplay, providing a novel form of game-music interaction.

I. INTRODUCTION

In recent years, the genre of music-based video games has attained widespread popularity. This sudden rise is due in part to the sophisticated processing capabilities provided by modern game console platforms, such as the Xbox 360 and PlayStation 3, which are capable of delivering rich graphics and innovative control interfaces that are tightly synchronized with real-time audio processing. Additionally, several titles in the music-based game genre, such as Guitar Hero, feature music composed and performed by well-known artists, which adds an element of popular culture for gamers and thus enhances the overall gameplay experience. It is clear that the importance of music in games is greater than ever and, in fact, the distinction between the gaming and music industries is blurring; while music is used to promote video games, video games are also used to promote music. Soundtracks from popular games can be purchased separately, and game studios have created music labels to promote this content.

At the same time, the use of the web as a gaming platform has increased significantly due to the wide availability of broadband connections, improved client processing power, and the capabilities afforded by Adobe Flash. Flash is the dominant platform for web-based game development since it allows programmers to author games on a cross-platform architecture and provides tools for easily implementing rich graphics, animation and user interface controls. Although Flash provides a straightforward means for deploying media-rich games on the web, its support for sound and music has been limited only to playback of pre-recorded clips. The lack of any buffer-based dynamic audio support in Flash has limited opportunities for developers to create gaming experiences relying on tight interaction with audio. Furthermore, ActionScript, Flash’s native development language, was never intended to accommodate computationally intensive algorithms, such as the signal processing required for real-time audio processing.

Recognizing the potential for developing audio- and music-centric games on the web, we have developed the Audio processing Library for Flash (ALF), which addresses the audio processing limitations of the Flash platform. ALF is based on Flash version 10 and capitalizes on the recently introduced Adobe Alchemy framework, which allows existing algorithms written in C/C++ to be compiled into byte code optimized for the ActionScript Virtual Machine for significantly improved performance [1]. By utilizing the dynamic audio capabilities recently added to Flash 10 and the computational benefits of Alchemy, ALF provides Flash developers with a library of common audio processing routines that can be incorporated into applications, such as reverberation, filtering and spectral analysis.

In adding real-time audio processing capabilities to Flash applications, ALF provides web games with an additional degree of sound interaction that has previously only been available on console or native PC platforms. Through the Alchemy framework, ALF is capable of supporting music-based games in Flash requiring responses from the player precisely timed to music. Other potential applications of ALF in web-based games include the addition of environmental sound processing to provide the player with a sense of direction and spatiality, resulting in a more immersive game world.

Although ALF can be used to enhance the audio of almost any Flash game, our goal is to enable a new paradigm of web-based gaming not only based upon the player’s interaction with audio, but actually driven by user-provided audio. This
potentially allows a player to choose a wide range of customized musical inputs, such as selections from their personal collection or completely user-generated music content (new recordings or perhaps remixes and mashups, which are becoming increasingly commonplace). Previously, tight coupling of game interaction with music for rhythm-based play has required significant development time and expertise. As we will demonstrate, ALF facilitates the development of games that are dynamically driven by the acoustic features of songs from a user’s music library, thus creating unique game play experiences depending on the provided audio content.

The remainder of the paper is structured as follows: In Section II, we present an overview of recent music video games, which incorporate sound directly within the gameplay. Section III briefly describes the development of ALF and how it can be implemented into existing Flash games. In Sections III and IV, we present the music- and sound-centric games we have developed, which demonstrate the utility of ALF for dynamic audio processing. Finally, we present our conclusions and discuss future work in Section VI.

II. BACKGROUND

A. Audio Processing for Rhythm-based Music Games

Currently, in most music-based games, the player’s objective is to follow the rhythm of the game’s music as precisely as possible using an external control interface. The emergence of these rhythm-based games has been due to the popularity of titles such as, Dance Dance Revolution, where players follow the tempo of game’s music by executing a prescribed set of dance maneuvers on a stage controller. The player’s performance in Dance Dance Revolution has little effect on the resulting audio aside from determining when the game ends if the player cannot keep pace with the dance maneuvers.

Guitar Hero and Rock Band have taken the concept of rhythm-based gaming a step further by utilizing audio processing to affect the game’s music based on the player’s interaction with an instrument controller. Whereas Dance Dance Revolution requires players to precisely “dance” in response to upcoming beats in the music, Guitar Hero and Rock Band require players to precisely “play” upcoming musical notes on their instrument controller. By successfully timing the playback of correct notes with the tempo of the game’s music, players can faithfully reproduce their instrument’s track in the game’s audio mixture and improve their score. If the player presses the wrong note or miscalculates the note’s timing, their instrument is degraded in the overall mixture. In terms of audio processing, the instrument tracks used in Guitar Hero and Rock Band are pre-recorded, and the user’s responses control when they are incorporated into the audio mix. These games also employ real-time audio processing by allowing players to add effects to their instruments, such as vibrato, delay and distortion.

The aforementioned titles have progressed the genre of music video games by providing an interactive and collaborative gaming experience centered on music. The common premise of these games, where score is based on a player’s skill in tracking the rhythm of the music, limits the ways in which players can creatively interact with the game [2]. Additionally, the audio tracks used to create the game music must be predetermined by the developer so that the rhythmic qualities can be extracted in advance. This creates extra work for the game developer since individual audio tracks must be obtained and analyzed for each instrument, and also limits the player to music the developer chooses. A system incorporating real-time audio processing may be able to reduce the offline analysis required to extract rhythmic cues and allow players to incorporate their own music collections into such games.

Microsoft’s Lips for the Xbox 360 is a rhythm-based karaoke game that requires players to precisely sing along to the lyrics of songs while scoring them in terms of pitch stability, rhythm, and vocal technique. Unlike Rock Band or other singing games, Lips allows players to incorporate their own music selections into the game by attempting to suppress the vocal components of the audio mixture so that players can sing along. The game is unable, however, to supply lyrics or evaluate vocal performance on player-supplied audio tracks, so they are only partially integrated into the game structure.

In a departure from other music games, Nintendo has developed Wii Music, which allows players to “play” musical instruments by performing appropriate gestures using the Wiimote controllers that simulate the physical actions required to play real instruments. In order to respond to player gesture, the game requires sophisticated audio processing capabilities. As a game, however, Wii Music lacks goal and score objectives and it is completely dependent on the Wii platform architecture.

B. Audio Processing for Games Based on User Supplied Content

While many rhythm-based music games limit the gamer’s interaction to pre-selected music, a small number of games have been designed to incorporate the rhythmic features of audio provided by the gamer.

Vib-Ribbon is a side-scrolling game developed for the original PlayStation console that bases the game control and environment on music the player supplies [3]. In terms of PlayStation games, Vib-Ribbon is unique in that the game loads into and plays directly from the console’s RAM, making the console’s CD-ROM drive available for the player to utilize their own music. The gameplay is similar to that of Guitar Hero and other rhythm based games in that it requires the user to tap a key that corresponds to a particular visual object and scrolls at a constant predetermined rate.

Audiosurf is a rhythm-based puzzle game developed for PC that allows the player to incorporate their own music library in order to drive gameplay [4]. Audiosurf employs preprocessing on player-selected music in order to generate game levels dependent on the dynamics of the audio. Being a PC game, it is easy for players to choose audio files directly from their personal digital music library, so the number of unique game levels is limited solely by the number of tracks in the player’s music collection. The objective is to collect blocks that appear on a track in time with the music’s rhythm. The track scrolls
at a predetermined rate and the player is able to move in one dimension perpendicular to the direction of the track.

AudioAsteroids is a simple game where the user avoids or destroys obstacles flying in space while collecting bonus objects. The properties of these objects are controlled by musical features, such as pitch and the number of simultaneous notes played. The pace of the game is determined by the extracted tempo of the song, and songs may be specified by the user [5]. Expanding upon the concept of AudioAsteroids, Briquolo is an open source version of the arcade game Breakout written in C++ for Windows and Linux. As in the original game, the objective is to eliminate a set of blocks in an enclosed area by bouncing a ball off a user-controlled paddle. The game has been modified to map features extracted from user-specified music files to parameters that affect the gameplay and graphics. The developers provide a default mapping of features to parameters, but also allow the user to define a mapping scheme in order to customize gameplay. Users choose from a fixed number of levels but may open any mp3 file from their local library to use as the background music for the level [6].

C. Music Video Games on the Web

In spite of the abundance and development of new music video games for console, PC and arcade platforms, few web games are centered around interactive audio processing. Music in Motion is a side-scrolling platform game developed using Flash that generates obstacles in synchrony with the game’s music. The music, however, is hard-coded by the developer, and the game is not capable of dynamic analysis of user-supplied content.

In developing ALF, our goal is to provide a framework and tools for Flash developers to enhance audio processing in existing games and to develop new, web-based games that utilize dynamic audio processing to generate unique and highly interactive game experiences.

III. ALF ARCHITECTURE AND FUNCTIONALITY

Prior to the release of version 10 in October 2008, Flash lacked support for dynamic (buffer-based) audio rendering. Development of custom computationally intensive methods, such as certain digital signal processing (DSP) algorithms, for Flash was not practical until the preview release of Adobe Alchemy (December, 2008) [7], [8]. Taking advantage of these new tools and features, we developed the ALF in order to provide web-based games with sophisticated audio processing functionality.

Prior to ALF, our solution for implementing computationally intensive DSP algorithms into our own applications involved using a hybrid Flash/Java architecture; Flash was used to implement the graphical user interface (GUI) and a “hidden” Java applet was developed to handle audio processing functions. Despite successfully implementing this architecture for our own games [9], we found that the complexity of interfacing two different platforms led to several problems, including a lack of error handling between Java and Flash and the inability to tightly synchronize GUI controls with the audio processing functionality. ALF is the result of our desire to equip Flash games with embedded and uninterrupted audio processing capabilities, without compromising the user’s gameplay experience.

A. Current ALF Implementation

Unlike previous versions, Flash 10 makes it possible to dynamically generate and output audio within the Flash framework. This functionality is asynchronous, allowing sound to play without blocking the main application thread. The Adobe Alchemy project allows C/C++ code to be directly compiled for the ActionScript Virtual Machine (AVM2), greatly increasing performance for computationally intensive processes. We have demonstrated significant performance gains using this system in a related paper [1]. With these tools, it is now possible to develop Flash-based applications that incorporate dynamic audio generation and playback capabilities without the need for an external interface for computation-intensive signal processing applications.

The Alchemy framework enables a relatively straightforward implementation of standard C code into Flash projects, so existing signal processing libraries written in C can be incorporated. C code is compiled by the Alchemy-supplied GNU Compiler Collection resulting in an SWC file, an archive containing a library of C functions, which is accessible in Flash via ActionScript function calls. An integrated application is created by simply including the SWC archive within the Flash project, producing a standard SWF (Flash executable) file when built. The Audio processing Library for Flash we have developed consists of a C-based library of methods wrapped in a SWC file that game developers can use for audio processing tasks in their games.

B. Example ALF Functions

Below are selected examples of common audio processing functions that are implemented in ALF. These functions have
application in audio analysis and information retrieval tasks [10].

1) getSpectrum: Computation of a sound’s frequency spectrum is used in many applications, such as creating visualizations from audio, and forms the core of many signal processing operations. Although the computeSpectrum function is available from ActionScript through the standard Flash library (an implementation of the Fast Fourier Transform, i.e. FFT, algorithm), it utilizes fixed values for several parameters, including the size of the transform, which determines frequency resolution. ALF’s getSpectrum method provides developers with greater control, allowing them to determine the desired DFT resolution as well as other parameters.

2) filter: For audio-centric applications, it is necessary to have filtering capabilities to achieve certain effects, such as frequency-based equalization (EQ). ALF’s filter function utilizes a fast, block-convolution method in the frequency domain to process an audio signal with a desired filter in an efficient manner. The filter type and its parameters are determined by the game developer.

3) reverb: The reverb function makes use of ALF’s filtering capabilities so that music and sound effects can be processed to simulate a desired acoustic environment. This effect can be used to enhance the game’s audio by giving the player a sense of physical space within the game, thus leading to a more realistic virtual environment and more immersive gameplay.

4) getIntensity: Intensity is a measure of the total energy in the sound and can be used to locate particularly important moments within music.

5) getBrightness: The distribution of energy across the frequency spectrum is strongly correlated with the perceived “brightness” of a sound. This value can be used to alter game environment variables in response to changes in timbre at varying locations within a song.

6) getFlux: Flux represents the amount of change in the spectrum over time. A large value corresponds to sudden changes in the audio, which can be used to drive events linked to sharp attacks in the musical texture.

7) getHarmonics: This function identifies individual frequency components within audio signals. These components can be used to generate additional sounds or audio effects. More detailed analysis of the harmonics can sometimes reveal information regarding the notes contained within the music and even the musical key.

C. Use of ALF for Novel Game Development

In order to demonstrate the potential for developing audio-intensive games using ALF and Adobe Flash, we provide examples of three games we have developed. The first example, Pulse, is a side-scrolling action game, which uses a player’s own music collection to define gameplay. The other examples, Tone Bender and Hide & Speak, are web-based, collaborative activities designed as educational games that employ rich interaction with sound to teach particular mathematical and acoustical concepts.

IV. Pulse

Pulse\(^1\) is a musically reactive, side-scrolling platform game originally developed for the web, but currently deployed using the Flash-compatible Adobe Integrated Runtime (AIR) environment for the desktop. Pulse resulted from a collaboration between the Music and Entertainment Technology and RePlay Labs at Drexel University with the goal of developing a unique game that utilizes a player’s personal music collection to drive gameplay. Unlike other music games, which rely on off-line audio analysis to determine the gaming environment, Pulse utilizes ALF functionality to update the game environment in real-time, mapping the quantitative features extracted from the audio to changes in the game’s environment variables. Real-time audio analysis enables Pulse to incorporate any music track specified by the user into the game, adding a strong element of personalization to the game. Ultimately, the audio-driven nature of Pulse increases the replay value of the game, since players aren’t restricted to scenarios based on a handful of pre-selected music tracks.

A. Gameplay Objectives

The player’s objective in Pulse is to traverse their character through a level while obtaining a maximum number of points before the music ceases playing. Players earn points based on the distance they advance through a level and the number of objects they collect along the way. Points are subtracted from the player’s score if they fail to advance to the next level or fail to avoid enemies along their path. As we detail below, the behavior of the game’s platforms, enemies and background graphics are determined by the features extracted from the music using ALF functions. The player is allowed to maneuver their character through the game by running, jumping or sliding. When jumping or sliding, the player’s character transforms into a “pulse”, which gives it the ability to defeat enemies upon contact. This dynamic, music-dependent environment poses a significant challenge for the player since movement must be carefully coordinated with the music in order to achieve the game’s objectives.

Pulse distinguishes itself from the other linear, rhythm-based games mentioned in Section II in several important ways. First of all, these games typically involve a screen that scrolls at a constant rate, such as the “virtual fretboard” used in Guitar Hero. The player is required to follow the rhythm of the music at a fixed rate, thus yielding a more predictable gameplay. Pulse differentiates itself in this regard by allowing the character to move freely in the 2-D space independent of the music. Game sessions in Pulse are also not restricted to the duration of the music as in other rhythm-based games. Instead, Pulse allows the player to determine the length of a session by building a custom music playlist so that the session continues uninterrupted until the playlist is concluded.

B. Dynamic Music Loading and Game Architecture

In developing Pulse, we sought to design a game architecture supporting cross-platform compatibility that could utilize

\(^1\)http://www.replay.drexel.edu/
audio tracks from the player’s digital music library. The former requirement made Adobe Flash an obvious choice, since it provides an environment suitable for rapidly developing and deploying applications on the web. However, a web-based architecture proved to be ill-suited for Pulse since security restrictions prevent Flash browser-based applications from easily accessing the client’s local file system.

Instead, AIR was used to implement Pulse as a desktop application able to access a player’s locally-stored music library. AIR utilizes the same integrated development environment as Flash, which permits use of the same tools and functions for implementing rich graphics and animation with ActionScript code. Also, like Flash, AIR applications enjoy the benefit of cross-platform compatibility since the runtime environment is available for all major operating systems (Windows, Mac, Linux).

The general game architecture of Pulse is illustrated in Figure 2, which depicts how audio analysis and playback is tightly synchronized with the video frame rate to update the game’s environment variables. After the player selects the desired track(s) from their music library, an ActionScript routine loads the track into memory. The game runs at a frame rate of 30 frames per second and analyzes the audio corresponding to the current video frame using ALF in order to extract acoustic features used to drive the game parameters. These features are returned to ActionScript in order to update the game environment attributes appropriately. It is important to note that as the audio frame is analyzed by ALF, it is also played back asynchronously without interruption. The process is repeated for each frame across the duration of a music track and each track in the playlist that defines the game session.

C. Driving Gameplay with Features Extracted via ALF

Pulse employs several ALF functions in order to dynamically shape the gameplay environment based on the current audio track. To illustrate their use, consider the screenshots of Pulse shown in Figures 3 and 4. The bottom screen shows a situation where audio is not being played, as encountered when the game is loading a track or after the completion of a level. Without audio, the game environment is empty and the objects and enemies are motionless. When the audio begins to play, Pulse utilizes the features of the music to add graphics and animation to the game environment.

Pulse maps features of the music extracted using ALF directly to game environment parameters. ALF provides several routines that describe the spectral content of music, which correlate to qualitative descriptions of audio (e.g. brightness and intensity). Since Pulse analyzes the game’s music in
short-time segments, ALF can efficiently extract the spectral features on a per-frame basis, allowing the game’s environment to be dynamically updated. The primary game environment variables that react to changes in the game’s audio include the background scenery, the player’s obstacles and collectibles as well as the platform supporting the player.

The getBrightness function is called in order to adjust the hue of the game’s background color (the initial background color for a level is determined by the genre of the music as extracted from the track’s metadata). This feature provides some control over the mood of the game session since the background color will represent the relative brightness at any point in the song. The level’s background color is also affected by the intensity of the audio being processed in order to provide a relative measure of loudness, obtained through the getIntensity function. This is mapped to the transparency parameter of the game’s background.

The behavior of the player’s collectibles and obstacles are dictated by the intensity and flux values derived from the music. The getIntensity function is used to control the size of the enemies and collectible coins so that they change in synchrony with the relative intensity of the audio. This adds a visual “pulsing” effect to the object and requires the player to precisely time their jumps in order to collect or avoid these objects appropriately. Additionally, sound intensity values exceeding a certain threshold will cause enemies to fire projectiles at the character. The game’s enemies move as dictated by the getFlux function, which provides an indication of how much the music changes over a short time period. The flux value is mapped to the enemy’s traveling velocity.

Another way in which Pulse utilizes audio to change the gameplay environment is by altering the slope of the platform supporting the player. The getIntensity function is again used to provide an indication of the music’s loudness, which is used to adjust the slope of the player’s platform so that increases and decreases in volume will require the player to traverse up and down the game path, respectively. These dynamic parameters require that players keep up with a constantly varying game trajectory, as dictated by the chosen music.

D. Metadata to Shape the Game Environment

While real-time audio analysis drives the gameplay, Pulse also makes use of the music file’s metadata in order to incorporate related media content so that the game can be visually enhanced in unique ways. Specifically, Pulse makes use of several API’s to extract images and lyrics from the web so that they can be incorporated into the current gaming session. By using Flickr’s ActionScript API, Pulse queries databases searching for images that are related to the artist, title or album name associated with the metadata of a particular song [11]. These images are incorporated into the game as background elements of the GUI. Pulse also utilizes an API provided by Lyricsfly, which allows the song’s lyrics to be queried based on its meta tags [12]. The lyrics are used to generate typographic “fireworks” in the game. As the player passes through checkpoints, words taken from the lyrics in the song are animated as an explosion, thus creating the visual effect of fireworks.

V. ALF IMPLEMENTATION IN OTHER FLASH-BASED GAMES

While the functionality of ALF is well-suited for single-player arcade-style games, it is also has applicability for other types of web games requiring sophisticated audio processing. In this section, we discuss web-based, collaborative games for education we developed in Adobe Flash that rely on ALF functionality. The purpose of these games is to serve as educational tools and platforms for collecting psychoacoustic data to help solve known problems in audio perception, namely the identification of musical instruments and the well-known “cocktail party” effect, which is described below.

Figures 5 and 6 illustrate how the game’s audio features correspond to object behavior in the game environment. In Figure 5, the player’s character is shown in an environment defined by a relatively static moment in the game’s music. The character is traveling along a surface with a modest gradient and is surrounded by very small enemies. However, during an intense moment in the music, as shown in 6, the gradient of the character’s surface has increased and the enemies, which were previously small and motionless, have grown in size and moved towards the player, emitting projectiles.
A. Tone Bender

*Tone Bender* was developed in order to explore perceptually salient factors in musical instrument identification [9]. The game requires that a player experiment with musical instrument sounds by modifying their timbre, in terms of the distribution of sound energy over time and frequency. These modified sounds are evaluated by many players to collect data regarding the perceptual relationship between modified acoustic features and their association with instrument identity.

1) Game Objectives: The game consists of a creation and listening interface, each with separate objectives that allow players to earn points.

In the creation interface, the player’s objective is to modify an instrument’s timbre as much as possible while still maintaining the identity of the instrument. The player can maximize their score by creating sounds near the boundaries of correct perception for an instrument, but ones that are still correctly identified by other players. Their potential score is based on the signal-to-noise ratio (SNR) calculated in terms of the deviation between the original and their modified instrument.

*Tone Bender*’s instrument creation interface features two windows, which allow the player to separately manipulate extracted parameters from the instrument, representing its amplitude and frequency characteristics. Figure 7 depicts the interface with the amplitude envelope display maximized, which allows the player to manipulate the instrument’s loudness over time by drawing their own curve with the mouse. The player can also switch the focus to the other representation, thereby maximizing the frequency characteristics, which allows the player to alter the spectral energy distribution by modifying the strengths of the instrument’s overtones.

The listening interface requires players to correctly identify an instrument, drawn from various instrument configurations submitted in the creation component. The player is allowed to listen to the sample instrument as many times as needed to determine the identity of the instrument they perceive. The spectral and amplitude characteristics for that sample are also displayed so that the player can utilize this information to help them make a choice, if desired. Points are awarded to the player based on the difficulty as judged by SNR of the modified instrument. By correctly identifying the type of instrument, the player receives the maximum number of points, while identifying only the correct family yields half the maximum number of points. No points are awarded if neither the instrument type or family matches those of the original instrument. The configuration of the listening interface is shown in Figure 8.

2) Audio Processing Functionality: *Tone Bender* makes use of multiple functions in ALF in order to drive the game audio and user interface. The getHarmonics function is used to extract the timbral parameters from the audio file, thus yielding the most dominant overtones contained in signal. These parameters are used within the game’s GUI so that the user can manipulate timbre by drawing out the desired loudness curve and/or modifying the spectral distribution by adjusting the overtones. Since *Tone Bender* requires the player to rapidly experiment with modified instrument signals, it is important that they receive immediate audio feedback to determine how their adjustments affect the resulting sound. As shown in Figure 9, *Tone Bender* accomplishes this by converting the screen coordinates from the GUI into physical parameters representing the instrument’s timbre. These parameters are used to efficiently generate individual audio buffers using ALF. Since Flash 10 permits buffer-based audio playback, sound is output without locking the user interface, which enables real-time interaction and feedback between the GUI and audio output.

B. Hide & Speak

*Hide & Speak* simulates an acoustic room environment to demonstrate the well-known “cocktail party” phenomenon, which is our ability to isolate a voice of interest from other sounds, essentially filtering out background sounds from an audio mixture [13]. The collaborative structure of the game was designed to collect evaluation data on the effects that source/listener positions and room reverberation have on speaker identity and speech intelligibility.

1) Game Objectives: As with *Tone Bender*, *Hide & Speak* consists of two components where the players have separate interfaces for creation and evaluation through listening.

In the creation activity, titled *Hide the Spy*, the player starts with a target voice and is instructed to alter the mixture of voices until the target voice (the “spy”) is barely intelligible.
within the mixture. The player can do this by adding more people to the room, increasing the reverberation, and changing the positions of the sources (including the listener position). Players can maximize their potential score by creating a difficult room where the target speaker is highly obscured, but still recognizable in the mixture. This difficulty is assessed by measuring the signal-to-interferers plus noise ratio (SINR) of the room. Audio for the target and interfering speakers are randomly drawn from the TIMIT speech database [14].

The Hide the Spy interface is shown in Figure 10, where a room configuration (20’ x 20’) is simulated as a 2-D space so that the player can easily visualize the speaker and source positions and correlate this with the resulting audio mixture. The room reverberation characteristics can be adjusted, and the interface allows the player to continuously listen to the room audio while adjusting the room’s parameters for immediate feedback.

The listening component of the game, Find the Spy, requires a player to determine if a target voice is present in a simulated room, where room configurations are drawn from those submitted from the creation component. The player is provided with an isolated sample of the target speaker’s voice (the “spy”) and the room audio mixture. When the target speaker is present in the room, they speak a different sentence from the one provided in the isolated sample, requiring the player to use the timbre of the target voice (as opposed to the speech content) to help isolate the speaker in the mixture. If the player correctly determines the target speaker’s presence in the room, they are awarded points based on the SINR of the room configuration. The Find The Spy interface is shown in Figure 11.

2) ALF Functions in Hide & Speak: Hide & Speak utilizes multiple ALF functions to generate the audio for the simulated acoustic room environment in Hide the Spy. Once the audio files of the target and interfering voices are loaded, they are processed by the getSpectrum function to generate spectral representations suitable for efficient filtering with the room’s reverberation characteristics. The room audio is generated by extracting the room parameters from the GUI and generating a reverb response for each person in the room with ALF’s reverb function. Each speaker’s short-time spectrum is filtered with their respective reverb characteristics, and these signals are then summed in order to generate the final room audio. This process is illustrated in Figure 12. Despite the large number of computations involved, the speed of ALF makes it possible to dynamically generate a room response for 9 speakers on a per-frame basis, allowing the player to manipulate speaker positions during playback and hear changes in real-time. Additionally, the process is implemented for each ear, taking into account the binaural differences in auditory perception to simulate a realistic room environment.
Additional information on the signal processing algorithms employed for Tone Bender and Hide & Speak is available in [1].

![Figure 12. Hide & Speak’s use of ALF functionality](image)

### VI. CONCLUSIONS AND FUTURE WORK

In this work, we have demonstrated how the use of our Audio processing Library for Flash enables greater functionality and flexibility when using sound in web-based games. By integrating ALF into their applications, developers are able to create responsive, interactive web game environments through dynamic audio processing. The parametric control provided by ALF expands the scope of projects that can be developed using Flash and ActionScript to levels of complexity previously attainable only on console or native PC platforms. As the popularity of music-based games continues to increase, we hope that ALF expands the developer’s creative palette, freeing them to investigate new directions and possibilities for the genre by taking advantage of user-provided and user-generated music content.

In addition to refining the current functions in ALF, we are working to add new functions. Although the getIntensity method provides a coarse measure of rhythmic activity, it does not provide an accurate detection of beats within music. We are in the process of implementing a real-time beat tracker that will extract beats and tempo information from the audio input signal. This would potentially provide an even greater level of synchrony between the visual and audio stimuli in Pulse and other games. We are also working to include additional audio effects, such as frequency-, amplitude-, and phase-modulation, chorus and flanging effects, and audio time-stretching and pitch-shifting, as well as a variety of methods for sound synthesis. As we continue to incorporate additional functionality, it is our hope that this library will elevate audio processing in Flash to be on par with its graphics and animation capabilities, while still retaining a similar ease of use for application developers.

We plan to release ALF as an open-source research project, which may be freely used by game developers and the research community. The current status of the project, including relevant documentation and source code, may be found at [http://schubert.ece.drexel.edu/research/ALF](http://schubert.ece.drexel.edu/research/ALF).

### ACKNOWLEDGMENT

This work is supported by NSF grants IIS-0644151, DRL-0733284, and DGE-0538476. The authors also thank other students in the Drexel Game Design Studio for their assistance in the development of Pulse: Nicholas Avalone, Thomas Bergamini, Evan Boucher, Nicholas Deimler, Kevin Hoffman, David Lally, Daniel Letarte, Le Tong, and Justin Wilcott.

### REFERENCES


