Auditory Image Model for iOS: a real-time mobile application

I. Project Overview

The Auditory Image Model (AIM) is a time-domain model of human auditory processing to represent how sounds are processed from the inner ear to the auditory nerve and cochlear nucleus. In comparison to the frequency-domain representation of sound using spectrogram, which is widely used today as a simple and crude approximation to human hearing, the stabilized auditory image generated from AIM more accurately simulates our perceived auditory image of complex sounds. This paper describes an iOS implementation of AIM, based on the latest C-version (AIM-C).

II. Background: Patterson’s Auditory Image Model

Roy Patterson and others at the Centre for the Neural Basis of Hearing, University of Cambridge, have been working on the Auditory Image Model. The “original” version was developed between 1985 and 1997, and was designed for Unix only. AIM was then implemented for C (aim2000) and Matlab (aim2003 & 2006). Since February of 2010, the “new” (and latest) AIM-C is being developed, primarily by Tom Walters, to simplify and enhance the design of the “legacy” AIM-C (2006-2010). News on the development of the new AIM-C can be found through http://aimc.acousticscale.org/.

Patterson, Allerhand, and Giguere (1995) give an overview of a software package for the auditory image model, which is designed with a modular architecture that provides the flexibility to choose between functional and physiological models for the three major stages of auditory processing. The first stage is spectral analysis, performed by either gammatone filtering or transmission line filtering, to represent the basilar membrane motion (BMM). The second stage is neural encoding, modeled by either two-dimensional adaptive thresholding (2D-AT) or inner haircell simulation, to yield neural activity pattern (NAP). The final stage is time-interval stabilization, implemented by either strobed temporal integration (STI) or autocorrelation, to generate an auditory image, stabilized auditory image (SAI) or correleogram. The modularized software implementation, which allows user to not only specify filter parameters but also choose models used in each of the three stages, makes it possible to combine models and to compare the images that result from their different combinations.

Patterson (2000) more thoroughly goes over the mechanism behind each of the stages used in the auditory image model, and does so by highlighting the model’s ability to adequately characterize elements of complex sounds as nosies, transients, and tones. This article demonstrates how auditory images, as a result of careful design of the models used to generate them, are able to characterize (and even separate out) noises, transients, and tones that make up complex sounds.

The Auditory Image Model has recently been explored to improve machine-hearing, using machine learning techniques to characterize sounds in ways that are more consistent to human auditory perception. For instance, Lyon, Rehn, Bengio, Walters, and Chechik, in “Sound retrieval and ranking using sparse auditory representations” (2010), describe using an auditory front-end (essentially process...
modules in AIM such as cochlea simulation, strobe detection, temporal integration, and stabilized auditory image) to generate sparse codes from audio, and to ultimately obtain feature vectors of audio documents.

III. AIM-i: Motivation, Goals, and Challenges

The main motivation for the AIM-i project is to make AIM more accessible and usable by making it available to iOS devices-- iPod Touch, iPhone, and iPad-- which have become quite popular and widespread in recent years. Upon beginning the project, the goal was to create a (quasi) real-time mobile application that renders SAI of the audio signal, input from the device’s microphone and, time-permitting, to offer additional features and user interfaces that modify various parameters affecting the graphical output.

In comparison to AIM-C, which is meant to be a complete, cross-platform software implementation of AIM in C++ (to run on a desktop or a laptop), AIM-i running on iOS mobile devices has a different use-case and implications. From a user’s point of view, having AIM-i in the form of an application is in some sense similar to having a disk image file, already compiled and ready to run. That is, users do not need to worry about downloading dependencies, configuration files, and tweaking make-files; they simply use a simple GUI to launch and interact with the app. From a developer’s point of view, writing for iOS would mean not having to worry about cross-platform compatibility issues, especially when it comes to grabbing the audio input from the device as well as rendering the graphics output on the device. However, there are also challenges involved with developing for iOS; two important issues are smaller processing power (when compared to a laptop or, even more so, a desktop machine) and reduced flexibility (such as limited OpenGL functionalities).

IV. AIM-i Organization

This section summarizes the hierarchy and organization of the files used in the AIM-i Xcode project. Files taken from AIM-C are in green, newly-written code that handles audio input is in blue, code that handles graphics output is in red, code serving as the “glue” with static variables and static function, runAIM(), is in orange, and code specific to the device UI is in purple. Folder and file names, as used in the Xcode project, are bolded; their descriptions are in regular font-weight inside parenthesis.

- **AIM** (Modules to build AIM and other files to support them)
  - Modules (All files under this directory subclasses the Module class)
    - Input (first module in the chain: takes device’s mic input)
      - ModuleiPhoneInput.h/.cc
    - BMM (basilar membrane motion)
      - ModuleGammatone.h/.cc
      - ModuleP2FC.h/.cc
    - NAP (neural activity pattern)
      - ModuleHCL.h/.cc
    - Strobes (determining strobe points for SAI)
      - ModuleLocalMax.h/.cc
      - ModuleParabola.h/.cc
    - SAI (stabilized auditory image)
      - ModuleSAI.h/.cc
    - SSI (size-shape image)
- ModuleSS1/.cc
  - Output (last module in the chain: renders graphics to device’s screen)
    - ModuleiPhoneOutput/.cc
  - Support (All files under this directory support the various modules above)
    - ERBTools/. (conversion between frequency and ERB)
    - SimpleIni/. (read/ write IO streams)
    - Common/. (primarily for logging)
    - Module/. (delineates modules with virtual functions to be defined)
    - Parameters/. (sets default and custom parameters for modules)
    - SignalBank/. (signal bank processed and passed between modules)
    - StrobeList/. (defines StrobePoint and a queue of StrobePoints)
    - linked- ptr/. (circular linked list.)
- MoMu (Mobile Music Toolkit, for iOS)
  - mo_audio/. (.mm (used to access the device’s audio buffer)
  - mo_def/. (#defines)
  - mo_thread/. (.mm (unused, but alternatives to x-thread)
  - x-def/. (#defines)
  - x-thread/. .cpp (concurrent processes: thread and mutex. used for runAIM() thread)
- OpenGL (OpenGL rendering on iOS)
  - EAGLView/. (.mm (view with OpenGL)
  - ES1Renderer/. (.mm (initializes all audio, starts AIM, renders graphics)
  - ESRenderer/. (super-class of ES1Renderer)
- Main View (UI: “front” view with AIM graphics)
  - MainView/. .mm
  - MainViewController/. .mm
- Flipside View (UI: “flip” view with AIM parameter settings)
  - FlipsideViewController/. .mm
- Application Delegate
  - AIMAppDelegate/. .mm
- Other Sources
  - globals/. .mm (contains Globals class with static global variables and AIM class)
  - AIM_i_Prefix.pch
  - main.mm
- Resources
  - Default.png, Default-Portrait.png (images displayed while app is loading)
  - aim- icon2.png (icon used in the device’s home screen)
  - FlipsideView.xib (graphical layout of UI elements for Flipside View)
  - MainView.xib (graphical layout of UI elements for Main View)
  - MainWindow.xib (container for MainView)
  - AIM-Info.plist (information property list)
- Frameworks
  - UIKit.framework
  - Foundation.framework
  - CoreGraphics.framework
  - AudioToolbox.framework
  - OpenGLES.framework
  - QuartzCore.framework
This flowchart below summarizes the main interaction between the various classes used in the project, using the same color-coding scheme described above. Important implementation details are presented in the next section.

V. AIM-i Implementation Details

In this section, essential code components are explained and presented with (simplified) code snippets taken from the project.

1) Handling Audio Input

The first part of audio involves accessing input audio buffer using MoMu’s MoAudio, and saving the buffer value in a global variable: `float*Globals::inputBuffer;`
The second part involves pushing values stored in `float* Globals::inputBuffer` to the chain of AIM modules; the code below belongs to the `ModuleiPhoneInput` module:

```c
void audioCallback( Float32* buffer, UInt32 numFrames, void* userData )
{
    Globals::effectiveFramesize = numFrames;
    for ( int i=0; i<numFrames; i++)
    {
        Globals::inputBuffer[i] = buffer[i*2];
        buffer[i*2+1] = 0;
    }
}
```

(2) **Handling Graphics Output**

The first part of graphics involves populating `vector<float*>` `Globals::outputV2`, a two-dimensional array, with `SignalBank` values output from the last AIM module (by default, the SAI module). This is performed in the `ModuleiPhoneOutput` module:

```c
void ModuleiPhoneOutput::Process( const SignalBank &bank )
{
    int n_channels = bank.channel_count();
    if( Globals::outputV2.size() != n_channels )
    {
        Globals::outputV2.clear();
        for( int i=0; i<n_channels; i++ )
        {
            Globals::outputV2.push_back( (float)bank.buffer_length() * 2 ] );
        }
       Globals::effectiveV2len = bank.buffer_length();
    }

    // initialize outputV2
    float inc = Globals::screenWidth / (float)Globals::effectiveV2len;
    for( int i = 0; i<n_channels; i++ )
    {
        float X = 0.0;
        for( int j = 0; j<bank.buffer_length(); j++ )
        {
            // x coordinate
            Globals::outputV2[i][2*j] = x;
            x += inc;
            // y coordinate
            Globals::outputV2[i][2*j+1] = bank[i][j] * Globals::verticalScale;
        }
    }
}
```

The second part is rendering `vector<float*>` `Globals::outputV2` using it as an OpenGL vertex
array. This is performed in `drawSAIOutput()`, called from `render()` of `ES1Renderer.mm`.

```c
void drawSAIOutput()
{
    int n_chan = Globals::outputV2.size();
    float height = Globals::screenHeight / n_chan;
    float y_offset;
    glLineWidth(1.0);
    for( int i=0; i < n_chan; i++ )
    {
        y_offset = height * i - Globals::screenHeight/2.0;
        glPushMatrix();
        glTranslatef(0, y_offset, 0);
        glVertexPointer(2, GL_FLOAT, 0, Globals::outputV2[i]);
        glColor4f(1.0 - 1.0 * i / n_chan, 1.0 - 1.0 * i / n_chan, (float)i / n_chan, 1.0f);
        glDrawArrays(GL_LINE_STRIP, 0, Globals::effectiveV2len);
        glPopMatrix();
    }
}
```

(3) The “glue”

`Globals.h/.mm` has two classes. First, the `Globals` class contains all static global variables that need to be accessed by multiple classes. Second, the `AIM` class contains a static function, `runAIM()`. This function creates all AIM modules, links them up based on user specifications, and begins processing. It runs as a separate, concurrent thread while the user is in `Main View`, and the thread is removed upon entering `Flipside View`.

```c
static void * runAIM( void * data ) {
    aimp::Parameters params, global_params;

    // Create all the modules.
    aimp::ModuleUIPhoneInput input(&params);
    aimp::ModulePZFC filterbank(&params);
    // aimp::ModuleGammatone filterbank(&params); // (use either PZFC or Gammatone)
    aimp::ModuleHCL nap(&params);
    aimp::ModuleLocalMax strobes(&params);
    aimp::ModuleSAI sai(&params);
    aimp::ModuleUIPhoneOutput output(&params);

    // Then link up all the modules into a chain that goes
    input.AddTarget(&filterbank);
    filterbank.AddTarget(&nap);
    nap.AddTarget(&strobes);
    strobes.AddTarget(&sai);
    sai.AddTarget(&output);

    aimp::SignalBank s;
    s.Initialize( Globals::numchannels, Globals::effectiveFramesize, Globals::srate);
    input.Initialize(s, &global_params);

    while (!input.done()) { input.Process(s); }
    return NULL;
}
```
4 User Interface (UI)

The following are screenshots of the Main View (rendering SAI) and the Filpside View.

VI. Evaluation

Overall, the pros offered by the iOS platform seem to outweigh the cons. Porting over the AIM-C code to iOS achieved a near real-time performance in generating an SAI for live audio mic input. Furthermore, having a user interface greatly mitigated the troubles of tweaking configuration files and re-building the project that one frequently has to go through while using AIM-C, making AIM-i more usable and accessible for the “non-specialists”.

I did notice a difference in performance between an iPad 3G and an iPhone. (According to Apple, an iPad has a “1GHz Apple A4 custom-designed, high-performance, low-power system-on-a-chip” processor; Apple does not reveal the processor power of an iPhone, but an unofficial source claims that an iPhone 3GS CPU runs at 412MHz.) As the computational power for mobile devices improve, I anticipate that the currently noticeable response-lag between input and output will become insignificant. As for usability, the larger screen size of the iPad offers greater room for exploration of the graphics output, although the iPhone’s smaller screen size does deter the user experience. Because the same project can be run on all iOS device types (iPhone Touch, iPhone, iPad), the application has great potentials for wide-spread use.

VII. Future Directions

This project has many areas for improvement. First, we need to ensure the application’s compatibility with the latest iOS version (iOS4); currently the code has been tested with devices running OS 3.x. Second, the graphics on the Main View could benefit from having labels (i.e. title, axis, tick-marks). Third, adding new features, such as being able to pause-and-play the output graphics, being able to load an existing sound file (as opposed to using live input from the mic), and being able to write the graphics output to a movie file for future reference, will all help to make this app more usable for casual-exploration of the AIM modules, as well as to perform additional analysis of sounds based on the AIM output.

1http://www.apple.com/ipad/specs/
2http://www.anandtech.com/show/2782
VIII. Acknowledgment

I would like to acknowledge Richard Lyon and Psych 303 “Human and Machine Hearing” course at Stanford University (Autumn 2010-2011) for instructions on AIM, and for general feedback and advice on this project. Special thanks to Tom Walters for code, documentations, and clarifications for the latest AIM-C, as well as for offering suggestions for AIM-i features. Finally, this project was made possible by the Music, Computing, and Design research group at CCRMA, Stanford University, through the Mobile Music Toolkit and iOS devices.

IX. References

AIM-C: a C++ Implementation of the Auditory Image Model

CNBH: The Auditory Image Model

MoMu: a Mobile Music Toolkit


