Developing Custom Audio Units

Asli Binal
Digitally Controlled Music Systems
May 2, 2005

Introduction

Using an application like Max/MSP was a great introductory tool for me to gain a better understanding of how signal processing works at a high level. Through creating a custom audio unit, I was able to dig further into manipulating the actual discrete samples to design an effect that would be have been trivial to implement in Max/MSP. This was a very useful exercise in learning about the complexities of audio programming. Using C++ and the Mac OS X Audio Toolbox environment, I developed a delay unit, which includes several parameters that can be modified in realtime. This document will illustrate how I approached the task of constructing an audio unit (AU).

Plug-Ins

In audio, plug-ins are software components that manipulate audio data and enable the creation of virtual instruments. Most digital audio workstations and sound editing tools come with the ability to insert a plug-in to enable an effect such as reverb, and like everything else, several standards exist. The most common plug-ins are VSTs (Steinberg), AUs (Apple), TDMs (DigiDesign) and LADSPA (Linux). These software components contain entry points, which are accessed by the audio workstation allowing the audio to be processed by the selected plug-ins. Developers can create their own plug-ins by using the appropriate development kits and APIs that are made available. One such environment is the Apple Core Audio development kit that includes a framework that manages the connections and allows the developer to focus on the signal processing functionality that they would like to implement. By facilitating the

creation of plug-ins through such frameworks, end-users such as researchers have a method for implementing effects that may not be available to them commercially.

Core Audio

The Mac OS X Core Audio environment consists of several components that enable audio programmers to gain access to the audio hardware abstraction layer (Audio HAL). Defined as the heart of the Mac OS X Core Audio system, HAL is responsible for all the housekeeping for ensuring that latencies are avoided, and that developers can easily access a method for controlling the input and output stream. Since the samples are delivered as floating point values, the programmer can design complex processing effects with a greater resolution. Perhaps the newest concept that initially caused some trepidation on my part was to figure out how single samples could be used in effects processing, for it is up to the developer to create buffers if that was required for the task at hand.

AU Framework

The AU Framework is a very useful tool for creating a new audio unit because it provides the programmer with a working environment that can be modified. A good place for beginners is to compile and install the default framework and to test it in a digital audio workstation or audio editing tool such as Peak. Once compiled, the framework will build a 'bundle', a file called *effect**.component***, which needs to be copied into either the user's ~/Library/Audio/Plug-Ins/Components folder, or the system's /Library/Audio/Plug-Ins/Components folder. This enables the component

manager to recognize any new components and make them available for audio applications. One of the lessons I learned was that the component manager does not scan for new .component files until the user logs out and logs back in, occasionally the component manager does not succeed, and this inconsistency can create some confusion at first. The AU mailing list has been a great resource because many of these issues are acknowledged, and the Apple programmers suggest solutions.

Challenges

One of the major challenges of this project has been the lack of useful documentation. The available resources are fragmented and don't necessarily answer all the questions that a developer may have. For example, the graphical user interface (GUI) layer for the default volume application has one slider and one text box but there is no explanation as to how the default GUI can be modified. In addition, the only sample AU that worked out of the box was not comprehensive enough to understand the complexities of audio processing, but this challenge enables the developer to come up with creative solutions.

An audio unit can only be called from a host application, therefore one cannot test an AU in debug mode, and this prompted me to create a custom logging utility to dump the parameter values.

AB_SimpleDelay

Using a circular buffer, which is the size of the sample to delay multiplied by the sampling rate creates the algorithm used in simple delay. This allows for an input sample to be saved into a buffer, and played back after a specified delay time along with the input and is a simple commonly used method of creating

delays. The user can specify the amount of wet/dry signal, which determines the amplitude of the original signal and the amplitude of the delayed signal to play back and the feedback gain determines how much of the output delay should be fed back into the input.

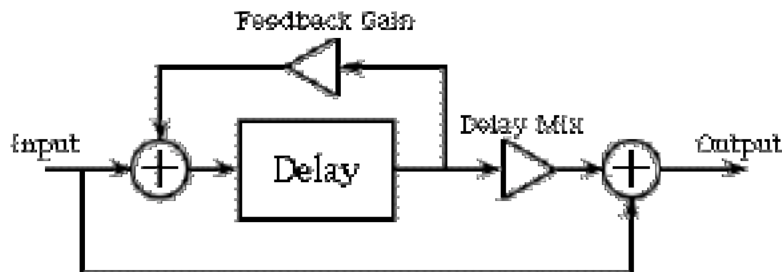

Figure 1: A simple delay with feedback gain. Image from http://www.harmony-central.com/Effects/Articles/Delay/

Here is a screenshot of the graphical user interface as it is displayed in Peak. There are sliders for the delay time, specified in seconds, a wet/dry percentage slider, and a slider that specifies the amount of feedback gain.

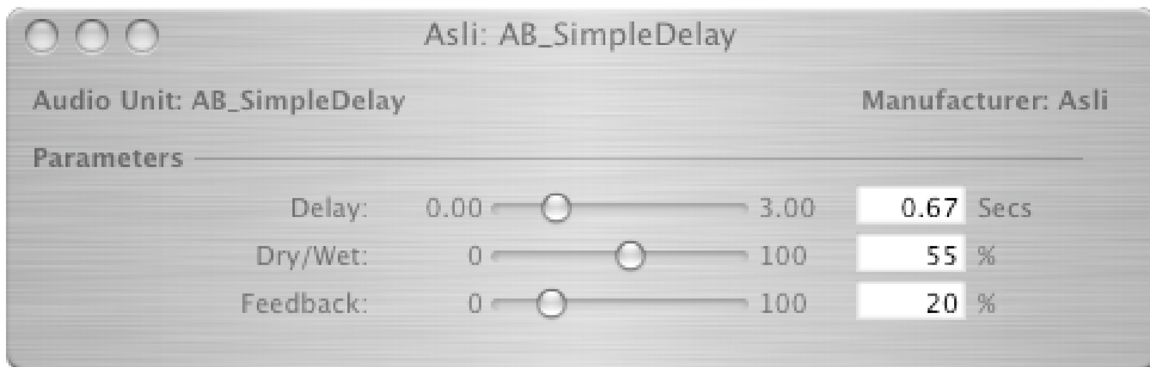

Figure 2: The user interface for Simple Delay, as displayed in Peak 4.0

To access the plug-in from a host application such as Peak, the user must insert the plug-in through the 'Plug-ins' menu. When the file plays back, the plug-in will be called through callback functions, enabling the effect to process the opened file.

Here is a screenshot of how to select my custom plug-in through Peak.

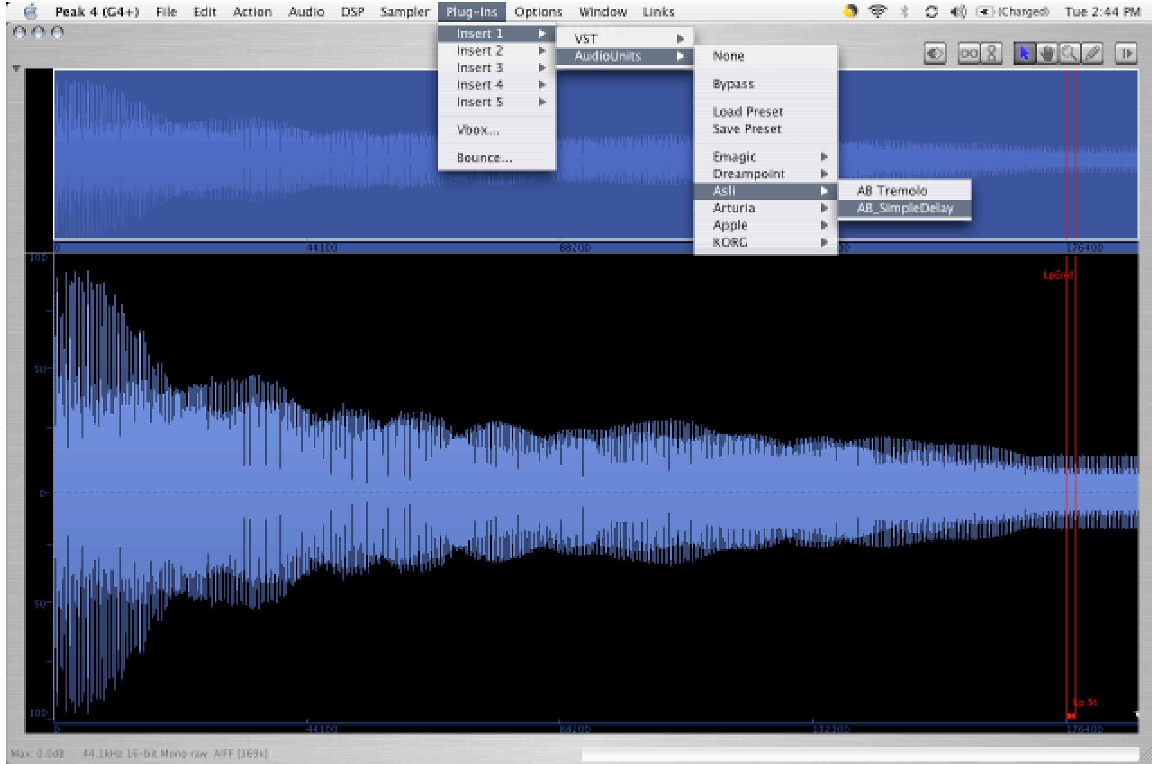

Figure 3: A Look at the C++ file that processes the signal

Instructions for installing AB_SimpleDelay

In order to test any new AU, there are a couple of steps that must be performed. In this section I will demonstrate these steps with the reason s behind the instructions.

1. After unzipping AB_SimpleDelay.zip, you will find a file called AB_SimpleDelay.component. This file is actually a bundle with the approprate files for executing the AU.
2. Place the .component file in /Library/Audio/Plug-Ins/Components directory.
3. Log out of your session, and log back in. This step is crucial as it enables the component manager to be executed so any new plug-ins that have been added will be recognized by the host application
4. Launch a host application such as Peak 4 or Logic and insert or select Plug-Ins. The audio plug-ins are listed by manufacturer, so my plug-ins will be located under "Asli", just as it appears in Figure 2.
5. Select the parameter values for the delay, and play back the original file which will be processed by the plug-in.

C++ Code Excerpts

The main portion of the signal processing is really taking place in two files. The most important files are AB_SimpleDelay.cpp and AB_SimpleDelay.h. In this section I will focus on the code, which is commented, to point out the sections that deal directly with the input signal and the processing.

In the following excerpt, I create a class that holds all of the variables and methods used in the processing task. The AUKernelBase class is the parent class for all audio unit signal-processing kernels. According to the documentation, each custom AU is required to subclass AUKernelBase for processing their stream.

The constructor creates a log file, and calls the initialization function to set states and retrieve parameter values:

```
class AB_SimpleDelayKernel : public AUKernelBase // most real work happens here
{
public:
     AB_SimpleDelayKernel(AUEffectBase *inAudioUnit ): AUKernelBase(inAudioUnit)
     {
          // create a log file for debugging purposes
          if((fLog = fopen("DelayLog.txt", "w+")) == NULL)
               return;
          // initialize states and parameters
          myInit();
     } ...
```

The destructor ensures that the log file is closed and the buffer is deleted so as to prevent memory leaks:

```
     virtual ~AB_SimpleDelayKernel(){
          //Free memory allocated during initialization
          delete [] delayLineStart;
          //Close the log file
          fclose(fLog);
     }...
```

Reset gets called by the host application every time the AU parameter(s) are changed, therefore initialization of buffers is necessary in order for the next execution to work properly:

```
     virtual void Reset()
     {
          fprintf(fLog, "RESET!\n");
          myInit();
     }
private:
     // Log file to help debug
     FILE *fLog;
     // delayTime is in milliseconds, amount of signal to feedback
     float delayTime, feedbackGain;
     // amount of wet/dry signal to play out
     float wetLevel, dryLevel, * outputSignal, * inputSignal;
     // delay buffer, and the output delay
     float * buffer, delayLineOutput;
     // pointers into the delay buffer, this wraps - circular buffer
     float * delayLineStart, * delayLineEnd, * readPtr;
     int  i;
};
```

In the AB_SimpleDelay.cpp file, all of the parameters are initialized and the delay buffer size is calculated and initialized. The delay buffer has to be large enough to hold samples from N seconds ago, therefore we multiply N by the sampling rate so that we have enough space to hold a total of Y samples:

```
// compute required buffer size for desired delay and allocate for it
int bufferLength = (int)(delayTime*GetSampleRate());
if(bufferLength <= 0)
  fprintf(fLog, "Delay buffer length in non-positive");

delayLineStart = new float[bufferLength];
if (delayLineStart == NULL)
  fprintf(fLog, "Couldn't allocate buffer in Delay");

//set up pointers for delay line
delayLineEnd = delayLineStart + bufferLength;
readPtr = delayLineStart;

// clear out the delay buffer that will be written to
do {
  *readPtr = (float)0.0;
}
while (++readPtr < delayLineEnd);

//reset read pointer to start of delayline
readPtr = delayLineStart;
...
```

Lastly, the output signal is calculated in the Process method. It is where I set the output signal to be the current signal + the delayed signal, ofcourse it isn't as simple as that, so let's look at the code:

Note: This is all happening in a while loop that iterates n number of times, where n is the window size. In this case, the host calls the process function with a buffer of 512 samples. The key points are that we set the delayLineOutput to read from the readPtr buffer, and that the readPtr is the buffer that gets updated, and wrapped once it reaches the end of the buffer size.

```
// get delayed sample
delayLineOutput = *readPtr;

// add the delayed signal to the input signal,
// ensure the correct amount of wet/dry is calculated
Float32 outputSample = dryLevel * inputSample + wetLevel * delayLineOutput;

// set the output sample, this gets passed back to host
*destP = outputSample;

// increment the counter based on number of channels
destP += inNumChannels;

// save input sample with any feedback to delayline
*readPtr = inputSample + feedbackGain * delayLineOutput;

//increment buffer index and wrap if necesary
if (++readPtr >= delayLineEnd)
  readPtr = delayLineStart;
```

Conclusion

Creating custom plug-ins is definitely a labor of love. Through the process you realize that documentation on this topic is minimal and scattered, but fortunately there enough developers posting to mailing lists and forums. Once you get past installing and ensuring that all of the components to your programming environment are compatible, it is rewarding to be able to manipulate a pure stream of samples. In the future, I would like to design AU plug-in instruments and feel that I could benefit from devoting some time to writing a beginner's guide to creating custom AUs.

References and Resources:

www.audio-units.com
http://willbenton.com/writings/audio-unit-effects.html
http://developer.apple.com/audio/coreaudio.html
http://www.harmonycentral.com