Bachelor's Thesis in Electronic Production and Design

Berklee College of Music

A Deep Look at Spectral Synthesis Techniques Through csConvolve

Project Summary

Darius Petermann

Supervisor: Dr. Richard Boulanger

March 2016

Berklee
College of Music

Contents

Introduction

The main purpose for this project report is to provide a brief overview of my undergraduate thesis, which aims at exploring different spectral synthesis and processing techniques, such as convolution, morphing and cross synthesis from a practical perspective. The outcome of the project resulted in an audio processing software on the iPad.

Motivation

This project was first and foremost motivated by the fact that convolution processing, when dealing with audio signals, is mostly renown for its power to change the acoustical property of a sound by multiplying its spectral content with the impulse response of any given acoustical environment. In other words, convolution is mostly used to simulate the reverberations of the sound in a given space.

Objectives

Through this project, I wanted to give myself the opportunity to explore the creative power of convolution, as well as other common spectral processing techniques - such as morphing and cross synthesis - in an less common way: as a synthesis technique.

Project Outline

The software uses Csound as its audio engine and allows for spectral manipulations, such as morphing, convolution and cross synthesis, given any sound source (real-time stream or stored files) as well as using any audio source as a filtering signal. Additionally, the software offers a set of spectral and time-domain pre/post-processing effects, which are described below:

Key Features: Instruments and Effects

csConvolve Synthesis/Processing Features		
Spectral Based	**Time-Domain Based**	**Misc.**
Convolution	Phase-Vocoder	Audio-Copy
Morphing	Buffer Glitch	Inter-App Audio
Spectral Blurr	Delay	FFT Plotting
Spectral Freeze	Distortion	
Spectral Cross (TBA)	Reverb	

Table 2: *List of all the processing and synthesis tools available as part of the software*

Programmatic Approach

This chapter aims to provide to the reader a concise overview of the approach that has been undertaken for this project. There are a plethora of things that could potentially be covered in such chapter, however we believed that providing an overview of the Csound implementation as well as the underlying spectral processing code was more than enough for a summary of that nature.

The Csound API

The architecture of the *Csound for iOS* API allows most processing to be defined directly from within *.csd* files, thus leaving very little processing left to be done inside the iOS audio callback function. The *CsoundObj* Objective-C class provides a convenient interface, which enables for UI bindings to, and from, Csound functionalities:

```
1    (void)addBinding:(id<CsoundBinding>)binding;
2    (void)removeBinding:(id<CsoundBinding>)binding;
```

Communication between the project and the API is established by setting output and input channel pointers:

```
- (MYFLT *)getInputChannelPtr:(NSString *)channelName
    channelType:(controlChannelType)channelType;
- (MYFLT *)getOutputChannelPtr:(NSString *)channelName
    channelType:(controlChannelType)channelType;
```

Notice that various data rate are supported for these pointers: *k-rate*, *a-rate* or *pvs-rate*. In our case, since we'll be mostly dealing with phase-vocoder streaming signals, we'd like to set these channels to *pvs-rate*.

The two following Objective-C Protocols methods (bindings and listeners) allow for data update to and from Csound:

```
- (void)updateValuesFromCsound;
- (void)updateValuesToCsound;
```

Finally, these two methods, defined in the *CsoundObj* interface, can be utilized when either a *.csd* file needs to be played, or a Csound score to be updated:

```
- (void)sendScore:(NSString *)score;
- (void)play:(NSString *)csdFilePath;
```

A Look at The Spectral Processors

The core processing functionality of the software is defined in a Csound file called *pvs.csd*, which contains the spectral processing set of instruments. *PVS* stands for *"phase vocoder streaming"*, an acronym that describes Opcodes that are particularly efficient in a streaming setting as they offer high audio quality and fast performance of spectral analysis and re-synthesis. The two orchestra instruments for offline convolution and morphing are defined respectively as follow:

```
inst 1

  ipsize  =    1024 ; partition size
  kdrywetmix    =    p6

  idel = (ksmps < ipsize ? ipsize + ksmps : ipsize)/sr

  aa, ab diskin p4, 1

  amono = a + b
  amono = amono * 0.15

  awet pconvolve amono, p5, ipartitionsize, 1

  adry1 delay aa, idel
  adry2 delay ab, idel

  afinal1 = (awet*kdrywetmix) + (adry1*(1-kdrywetmix))
  afinal2 = (awet*kdrywetmix) + (adry2*(1-kdrywetmix))

  aclip1 clip afinal1, 0, 0dbfs
  aclip2 clip afinal2, 0, 0dbfs

  outs aclip1, aclip2

endin
```

Code Sample 1: *Implementation of convolution in Csound*

Code Sample 1 Comments

In the Code Sample 1 - line 5, we notice that the latency of the convolution Opcode can be computed as the sum of the block size and the partition size, if the block size happens to be smaller than the partition size. Otherwise the latency would simply be equal to the partition size. This computation [1] is crucial in order to compensate the time-offset that the convolution operation will produce between the dry and wet signal. As a result, we can then delay the dry signal by this same exact latency amount (line 14-15).

```
inst 2

  iN              = 1024 ; fft size
  iH              = ifftsize / 4 ; overlap
  iW              = ifftsize ; window size
  iwintype        = 1 ; von-Hann
  kampint         = p6
  kfrqint         = (1 - p6)
  Sfile1          = p4
  Sfile2          = p5

  aa, ab          soundin         Sfile1
  ac, ad          soundin         Sfile2

  ain1 = (aa + ab) / 2
  ain2 = (ac + ad) / 2

  fftin1    pvsanal ain1, iN, iH, iW, iwintype
  fftin2    pvsanal ain2, iN, iH, iW, iwintype
  fftmorph pvsmorph fftin1, fftin2, kampint, kfrqint

  aout      pvsynth      fftmorph

  outs (aout * .5), (aout * .5)

endin
```

Code Sample 2: *Implementation of spectral morphing in Csound*

Code Sample 2 Comment

In the Code Sample 2 - line 3-6, it is important to denote how all the values related to spectral analysis and synthesis (*N, H, W, Window Type*) are set to constant values. This could be easily improved by making these values dynamic instead: we could provide the most optimal values based on the nature of the sounds being processed. For the sake of simplicity, we opted to provide generic values that work fairly well in any given context.

The core of the process occurs within line 18-22: here, we first analyse the two given sounds using the spectral analysis Opcode *pvsanal* [2]. This Opcode will output an *fsig* signal by performing a phase-vocoder overlapp-add analysis of the sound. Once the two *fsig* signals are computed we can feed them to the morphing Opcode *pvsmorph* (line 20) [3]. *pvsmorph* will interpolate between the frequency and amplitude components of two given sounds (through their respective *fsig* representations). Finally, we call the *pvsynth* [4] Opcode in order to synthesize our morphed *fsig* signal back into a time-domain representation.

Conclusion

Outcome and Future Work

The initial idea behind this project meant to put a special emphasis on spectral transformations, specifically achieved in a streaming context, Unfortunately, due time constraints, the focus on the streaming aspect of the signals had to be put-aside. Consequently, there is still room for optimizations to be brought to the current implementation (as mentioned in the Programmatic Approach section), which is currently specifically targeting offline processing yet works fairly well in a streaming setting.

With the introduction a few years ago of the Csound Cuda Opcodes [5], which enables for the performance of heavy processes, such as the sliding phase vocoder [6], on the GPU hardware, we believe that latency could be reduced substantially in a way such as that even heavier spectral processes could be achieved [7].

A short "hands-on" video demonstrating the software capabilities can be found at the following [8].

Bibliography

[1] Ingalls, M. pconvolve opcode. *Csound Floss Manuals* (2004).

[2] Dobson, R. pvsanal opcode. *Csound Floss Manuals* (2001).

[3] Lazzarini, V. pvsmorph opcode. *Csound Floss Manuals* (2007).

[4] Dobson, R. pvsynth opcode. *Csound Floss Manuals* (2001).

[5] ffitch, J. What's new in csound 6.06. *Csound* (2015).

[6] Bradford R., f. J., Dobson R. Sliding is smoother than jumping (2005).

[7] Bradford R., f. J., Dobson R. The sliding phase vocoder (2007).

[8] Petermann, D. Ep491 final demo video. URL `https://www.youtube.com/watch?v=TnYs7vVqut0`.