AUDIO PROCESSING WITH PROCESSING
BUILDING THE ECHO FILTER

SANDER IN 'T VELD AND JAN WESTERDIEP

1. Introduction

The idea of this problem is the following: you have a sound which you want to play back including a set lag of itself. See it as a canon in two voices. To solve this problem in Processing, we will be using a Cyclic Buffer and implement the actual Echo Filter as an `AudioEffect`. Some knowledge of sound buffers will be presumed in the following article.

2. The Cyclic Buffer

Buffers are arrays of samples with fixed length. We imagine this buffer to be 'cyclic', i.e. that the last index precedes the first. To make this buffer cyclic, we will have to keep count of the last inserted sample index. This makes for the following class outline.

```
<<CyclicBuffer>>=
  class CyclicBuffer {
    private float[] buffer;
    private int pos;

    CyclicBuffer (float len) {
      buffer = new float[len];
    }

    <<CyclicBufferMethods>>
  }
```

Every next sample will then be inserted directly behind the last. To facilitate this, we use the `pos` variable. To make sure out array index never gets out of bounds, we take the modulus of `pos`.

```
<<CyclicBufferAddSample>>=
  public void addSample(float sample) {
    pos = ++pos % buffer.length;
    buffer[pos] = v;
  }
```

To take samples out, we construct a method that gets samples relative to the last inserted sample. This is because we don't see the CyclicBuffer as an array with first or last incides, but rather a head and a tail. To account for the way Java implements modulo, we have to make sure `pos-i-1` is greater than (or equal to) zero.

Date: June 18, 2012.

```
<<CyclicBufferMethods>> =
  <<CyclicBufferAddSample>>

  public float getSample(int i) {
    while (pos-i < 0) {
      i -= buffer.length;
    }
    return buffer[(pos-i) % buffer.length];
  }
```

3. Echo Filter

As previously stated, we will be implementing EchoFilter as an AudioEffect, from which we can deduce needing the mono and stereo `process` methods. For convenience, the class constructor will take two parameters; a `float sampleRate`, which will usually be the 44.1KHz used in digital audio, and a `float tdelay`, which of course stands for the time between the original sample and the echo of this particular sample. With `sampleRate` samples per second, and `tdelay` seconds delay, we will obviously have a `sdelay = sampleRate * tdelay` sample delay. Our left and right CircularBuffers will then contain `sdelay` elements each.

```
<<EchoFilter>> =
  public class EchoFilter implements AudioEffect {
    CyclicBuffer leftB;
    CyclicBuffer rightB;

    EchoFilter (float sampleRate, float tdelay) {
      int sdelay = (int) floor(sampleRate * tdelay);
      leftB = new CyclicBuffer(sdelay);
      rightB = new CyclicBuffer(sdelay);
    }

    <<EchoFilterProcessMono>>
    <<EchoFilterProcessStereo>>
  }
```

In mono sound, both speakers emit the exact same samples. This means that our stereo `process` method will contain more or less the same information the mono variant will, only twice for each speaker[1].

We made our CircularBuffer of `sdelay` length for two reasons: first, we don't need to store any more information after the echo was pushed, and secondly, this makes for a beautiful and intuitive `process` method. This is because `CircularBuffer::getSample(-1)` will return us the 'tail' of the buffer, i.e. the element that was added the longest ago. To ensure every sample will still be in the interval $[-1, 1]$, we'll take the average of both the existing sample and the echo[2].

```
<<EchoFilterProcessMono>> =
  public void process (float[] signal) {
```

[1]We unfortunately can't directly call the mono variant twice for each speaker in the implementation of the stereo variant because we'll be using two different sound buffers.

[2]In the event you'll want to make the echo sound more 'echo'-y, be sure to make the echo less loud.

```
    for (int i = 0; i < signal.length; i++) {
      leftB.addSample(signal[i]);
      signal[i] = (leftB.getSample(-1) + signal[i])/2;
    }
  }
```

```
<<EchoFilterProcessStereo>> =
  public void process (float[] left, float[] right) {
    for (int i = 0; i < left.length; i++) {
      leftB.addSample(left[i]);
      left[i] = (leftB.getSample(-1) + left[i])/2;
    }
    for (int i = 0; i < right.length; i++) {
      rightB.addSample(right[i]);
      right[i] = (rightB.getSample(-1) + right[i])/2;
    }
  }
```

4. Processing Sketch

To glue it all together, we'll make the final sketch as follows. This pretty much speaks for itself: we instantiate a simple window with default settings, load a file, loop it and attach our newly created EchoFilter effect.

```
<<EchoFilterSketch>> =
  import ddf.minim.*;

  void setup() {
    size(200,200);

    Minim minim = new Minim(this);

    AudioPlayer player = minim.loadFile("lvl_eagle.mp3");
    player.loop();
    AudioEffect echo = new EchoFilter(player.getFormat().getSampleRate(), 1);
    player.addEffect(echo);
  }

  <<CircularBuffer>>
  <<EchoFilter>>
```

5. Discussion

In the writing of this article, a lot of code has been rewritten and refactored along the way. The first hurdle was of course the implementation of the actual echo. At first we chose to implement it as a listener/signal combination, but after reading the assignment the `AudioEffect` interface made much more sense.

Another point of interest was the handling of the position. Ultimately we chose to increase the `pos` variable *before* every sample add as opposed to *after*. This was to shave a few bits of code (and increase legibility) in the `getSample` method[3].

To account for the stereo speakers, we had to in a sense repeat ourselves by effectively retyping what the mono method contained. To solve this, we would need to add an extra class variable `activeBuffer` and make sure the appropriate buffer is read/written. This would require an `if`-statement inside our mono `process` method[4] resulting in more cluttering. We ultimately decided against it.

In conclusion we can say that this code quite effectively resolves our initial problem.

Appendix A. Java code

```
hw3_2.pde:
  import ddf.minim.*;

  void setup() {
    size(200,200);

    Minim minim = new Minim(this);

    AudioPlayer player = minim.loadFile("lvl_eagle.mp3");
    player.loop();
    AudioEffect echo = new EchoFilter(player.getFormat().getSampleRate(), 1);
    player.addEffect(echo);
  }

_CyclicBuffer.pde:
  class CyclicBuffer {

    private float[] buffer;
    private int pos;

    CyclicBuffer(int len) {
      buffer = new float[len];
    }

    public void addSample(float v) {
      pos = ++pos % buffer.length;
      buffer[pos] = v;
    }

    public float getSample(int i) {
      while (pos-i < 0) {
        i -= buffer.length;
      }
```

[3]The original code was substituting `pos-i-1` for every `pos-i`.

[4]With a simple pointer, this would of course all be unnecessary. Too bad this isn't C.

```
    return buffer[(pos-i) % buffer.length];
  }
 }

_EchoFilter.pde:
 public class EchoFilter implements AudioEffect {

  CyclicBuffer leftB;
  CyclicBuffer rightB;

  EchoFilter (float sampleRate, float tdelay) {
    int sdelay = (int) floor(sampleRate * tdelay);
    leftB = new CyclicBuffer(sdelay);
    rightB = new CyclicBuffer(sdelay);
  }

  public void process (float[] signal) {
    for (int i = 0; i < signal.length; i++) {
      leftB.addSample(signal[i]);
      signal[i] = (leftB.getSample(-1) + signal[i])/2;
  }
 }

  public void process (float[] left, float[] right) {
    for (int i = 0; i < left.length; i++) {
      leftB.addSample(left[i]);
      left[i] = (leftB.getSample(-1) + left[i])/2;
    }
    for (int i = 0; i < right.length; i++) {
      rightB.addSample(right[i]);
      right[i] = (rightB.getSample(-1) + right[i])/2;
    }
  }
 }
```