Music 421 Final Project : This is not a comb filter

Jorge Herrera

03/16/2011

1 Introduction

This is not a comb filter is my final project for `MUSIC 421` (Audio Applications of the Fast Fourier Transform). It is an image driven real-time audio filtering application for iOS, inspired in Magritte's painting *This is not a pipe*[1].

The user is presented with an image (by default the one showed in Figure 1). When two fingers are place onto the image, the pixel values along the line defined by the two fingers are used as the sampled magnitude response of a time-varying filter that processes the microphone input in real-time. As the user moves the fingers on the screen, the filter's magnitude response changes.

The user change the "source" image either by browsing the image library on the phone, by browsing the internet, or by taking a picture using the integrated camera.

Figure 1: Application's user interface

2 Design

Figure 2 shows a block diagram of the most relevant parts in the system, identifying the different processing steps and the interaction between visual and audio systems.

2.1 Image processing

Since this is an image processing is not the central of the present writeup, I will briefly point out one key aspect.

The main problem to solve in this step is to accurately sample the image to generate enough information to derive the filter's spectral envelope while updating the line channel information

[1] see `http://en.wikipedia.org/wiki/The_Treachery_of_Images`

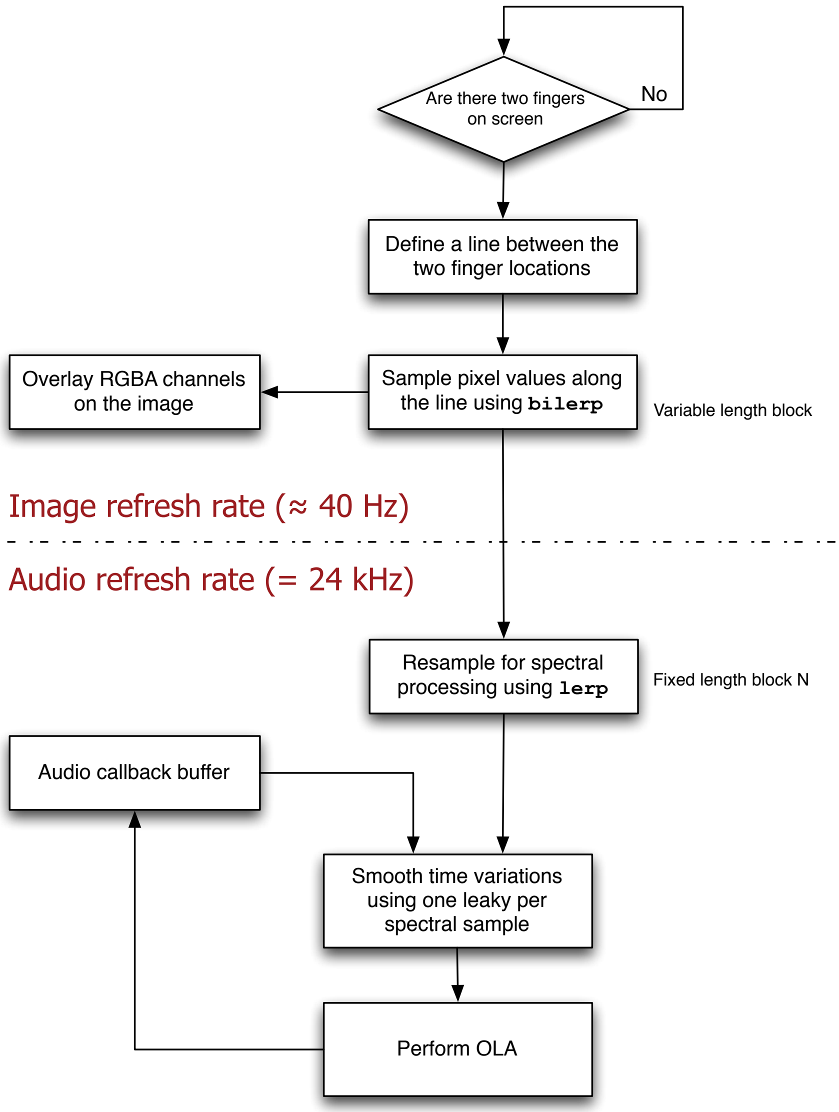

Figure 2: System overview

visually on screen in real-time, providing a coherent audiovisual feedback. Note that visual and audio systems operate at different sampling/refreshing rates. Therefore, the impose different resolution constraints.

I decided to keep a constant "visual resolution" in this image processing step. In other words, to sample the image at intervals of pixel on the screen. This is easy when the line is aligned exactly vertically or horizontally, but it requires interpolation in any other orientation. Bilinear interpolation (`bilerp`) was used to derive these values. While sufficient for visual accuracy, this sampling is not necessarily enough for audio processing purposes. I'll address this problem later.

Is also important to note that four visual channels (RGBA) are interpolated simultaneously. In the current implementation only of the application, only one of the channels –the red one– is being use to drive the filter.

2.2 Audio processing

To implement a real-time filtering in the spectral domain a STFT is required. There are at least two options to implement the STFT: 1) Overlap And Add (OLA); 2) Filter Bank Summation (FBS). Although originally I was planning to implement the FBS solution, I decided to switch to an OLA design because of the time-varying nature of the filter.

As mentioned before, the Spectral Envelope derived in the image processing step does not necessarily have enough frequency resolution for the audio processing. In particular, for a N long FFT we need $N/2$ spectral envelope values. That's why is necessary to implement a second interpolation of the "visually interpolated" line. A simple linear interpolation (`lerp`) was selected for this purpose.

Other design decisions/restrictions made to keep things simple in a first iteration of the application are:

1. A fix overlap of 50% in this first implementation
2. Parametrized FFT length (currently 4 times the window length)
3. Fixed linear frequency mapping
4. Zero-phase filter assumption
5. Monophonic processing

Most of these restrictions are very easy to modify/relax, to make the application more interesting and increase the audio quality.

3 Implementation

3.1 UI and Image processing

For the image processing, the display of both the image and the 4 channel lines was done in C++/OpenGL. The rest of the visual system was done using the iOS SDK/Objective-C/Interface Builder, with the help of an open source code found at `vocaro.com`[2].

Figure 3 presents two examples of the visual feedback provided to the user. Is possible to see 4 colored lines representing the corresponding RGBA channel (alpha channel is represented in white). The ref "profile" is being use to derive the spectral envelope of the filter.

[2]see `http://vocaro.com/trevor/blog/2009/10/12/resize-a-uiimage-the-right-way/`

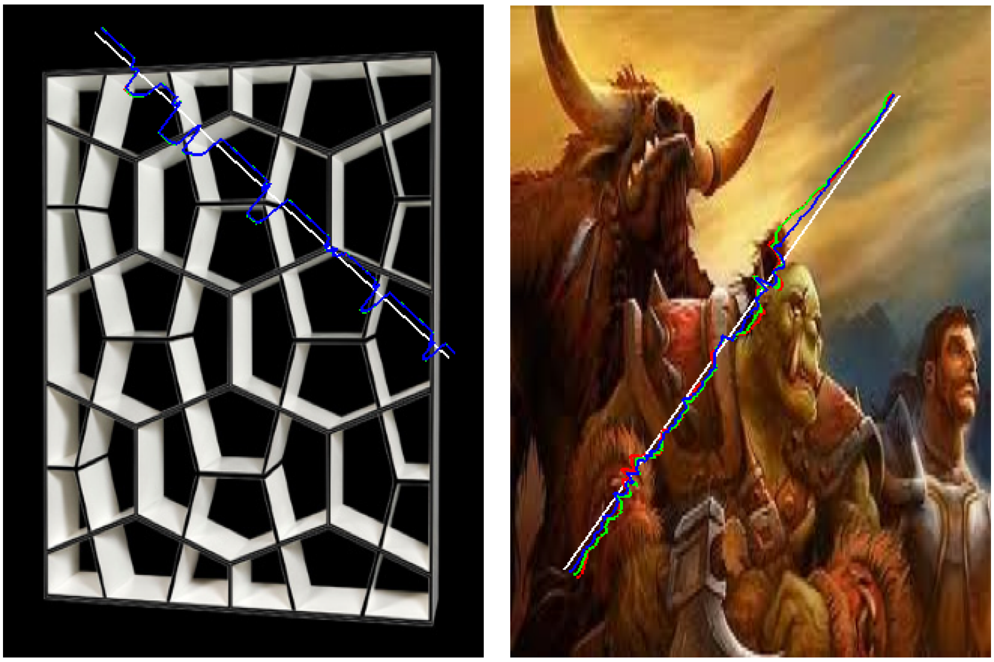

Figure 3: Two screenshots of the application, using different images to drive the filter

3.2 Audio processing

The Mobile Music toolkit[3] takes care of the low level audio routing, simplifying the interaction between graphics, audio and touchscreen interactions.

The core of the audio processing is the OLA implementation, which was encapsulated in its own class (see Appendix 5.1).

Since the audio callback provides contiguous non-overlaying blocks, the OLA class needs to split the M–samples long input block into two, in order to perform the 50% overlap. The first half is combined with the the second half of the previous audio input block and the processed. The returned values are added according to the OLA procedure and the first half (R samples) of the output audio bock are computed. The second R samples of the output audio block are computed by spectrally processing the whole current input block and performing the add step. Finally, the second half of the input block is saved for the next iteration.

The processing step just mentioned is as follows: First, the block is windowed, zero–padded and its forward FFT is calculated; then, the spectral envelope is smoothed over time using one leaky integrator per spectral bin; the "time–smoothed envelope" multiplies, sample by sample, the FFT of the input signal; finally, the inverse FFT is performed and the output is returned, to bused in the addition step.

The integration between image processing and audio processing is simple. Every time the line changes–every time time the user moves one or two fingers–a new spectral envelope is computed. The visual interpolation of the red channel is resampled get $N/2$ magnitude values. This defines a "target envelope" that drives the leaky integrators mentioned in the previous paragraph.

[3]see `http://momu.stanford.edu/`

The main parameters used in the application are:

- Sampling frequency (F_s): 24.000 Hz
- Window length (M): 512
- Overlapp length (R): 256
- FFT length (N): 4096 (4*M)

Although optimally is required that $F_s \geq 44000$, the selected value of F_s provides a good trade–off between audio quality and real-time performance.

4 Conclusions and Future Work

This first iteration of the application successfully completed the audio/visual integration, providing a real-time, image driven filtering of the microphone input in an iPhone.

Although all the parts work together, many improvements could be made to improve the quality and usability of the application. Some ideas are:

- Implement minimum phase filter (instead of the zero–phase one)
- Combine different channels in interesting ways (e.g. Red channel mapped Left channel and Blue channel mapped to Right channel)
- Change frequency to a perceptual scale (e.g. BARK scale)
- Parametrize overlap % and windows
- Image carrousel to select images

5 Appendix

5.1 Relevant source code

```
/*
 *  OLA.h
 *  CombSynth
 *
 *  Created by Jorge Herrera on 3/15/11.
 *  Copyright 2011 Stanford. All rights reserved.
 *
 */

/*

 Based on SASP, p. 231

*/

#ifndef __OLA_H__
#define __OLA_H__

#import "mo_fft.h"

#define RHO 0.5  // leaky integrator constant


// implements 50% OLA using a Hamming window
class OLA {

// public methods
public:
  /*
   Constructor:
   Performe some sanity checking, initializes some instance variables and
   allocates memory for the different arrays
   */
  OLA( unsigned int M, unsigned int N );

  /*
   Destructor:
   Free up memory used by the instance arrays
   */
  ~OLA();

  /*
   processBlock:
   Does all the processing related to an stereo input buffer. For now it
   converts the input into a mono signal (takes only the left channel)

   Arguments:
    block:     input block, interleaving left and right samples (the same
               format received from the audio_callback)
    numFrames: number of samples in a channel
```

```
     output:    array of floats to hold the output block associated with
                the input block
    */
  void processBlock( float * block, unsigned int num_frames, float * output );

  /*
   When the user changes the finger placing in the image, the target spectral
   envelope changes accordingly
   */
  void updateSpectralEnvelope( float * new_envelope, unsigned int num_points );


// private methods
private:
  /*
   Helper method that takes a monophonic input and returns the corresponding
   output, after performing the spectral processing. It takes care of the
   actual OLA step and related book keeping.

   Arguments:
   input:    N samples long monophonic signal. The first M samples are
   considered. The rest is a place holder for zero padding
   first and later it will contain required time-domain samples
   to be used in the OLA step.
   output:    M/2 samples long array to return the processed block
   (M/2 is because of the fix 50% overlap)
   */
  void processOverlappedFrame( float * input, float * output );

  /*
   Helper method to perform the spectral modification on a block

   Taks a N samples long time domain input (the first M samples being actual
   samples and the rest is a place holder for zero padding).

   Performs the following operations:

   1. Apply the window to the first M samples
   2. Zeropad the rest of the input signal
   3. Computes a forward FFT (in place)
   4. Applies spectral modifications
   5. Computes the inverse FFT (in place)

   All N returned samples must be used in the OLA step

   */
  void spectralProcess( float * input );



// private attributes
private:
  unsigned int M;      // Window size (in samples)
```

```
  unsigned int N;       // FFT size
  unsigned int Nover2; // Half FFT size
  unsigned int R;       // Number of overlapping samples (given that I'm
                        // using a fix 50% overlap, R = M/2)

  float * previous_half;  // Previous block required to perform "block
                          // decomposition" required for the overlapp
  float * window;         // Actual window used per processed block
  float * ola_buffer;     // Circular buffer to perform the OLA
  float * current_block;  // Used to "construct" the overlapped time domain
                          // block before going into the freq. domain

  unsigned int ola_buffer_head;  // Needed to implement the circula buffer

  float * spectral_envelope;  // Spectral envelope (derived from the image)
                              // to be imposed onto the audio signal
  float * last_envelope;      // Last envelope (used to smooth out the changes
                              // in the spectral envelope, using a
                              // Leaky-integrator)

  float a1, b0;  // leaky integrator constants used to smooth out spectral
                 // envelope changes

};


#endif  // end __OLA_H__
```

```
/*
 *  OLA.mm
 *  CombSynth
 *
 *  Created by Jorge Herrera on 3/15/11.
 *  Copyright 2011 Stanford. All rights reserved.
 *
 */


#ifndef __OLA_MM__
#define __OLA_MM__

#import "OLA.h"
#import "utils.h"

OLA::OLA( unsigned int M, unsigned int N ) : M(M), N(N), R(M/2) {

    assert( M && !(M & (M - 1)) );    // fail if M is not a power of 2

    // if N is not a power of two, make it a power of two
    if (!( N && !(N & (N - 1)) )) {
        printf("N = %d is not a power of 2\n", N);
        float l2 = log( N ) / log( 2 ); // compute the log2(N)
        N = 1 << (int)ceil( l2 );
        printf("making N = %d\n", N);
```

```
        this->N = N;
    }

    Nover2 = N/2;

    int i;

    // compute the hanning window to use
    window = (float *)malloc( M * sizeof(float) );
    MoFFT::hamming( window, M );
    for (i = 0; i < M; i++) window[i] /= 1.08; // compensate for the 1.08 COLA
                                               // constant of the hamming window

    // allocate memory for the different arrays used in the OLA process
    previous_half = (float *)malloc( R * sizeof(float) );
    ola_buffer = (float *)malloc( N * sizeof(float) );
    current_block = (float *)malloc( N * sizeof(float) );
    ola_buffer_head = 0;


    // initilize necessary arrays
    for (i = 0; i < R; i++) previous_half[i] = 0.f;
    for (i = 0; i < N; i++) ola_buffer[i] = 0.f;

    // allocate memory for the spectral envelope and related array and
    // initialize them
    spectral_envelope = (float *)malloc( Nover2 * sizeof(float) );
    last_envelope = (float *)malloc( Nover2 * sizeof(float) );
    for (i = 0; i < Nover2; i++) {
        spectral_envelope[i] = last_envelope[i] = 0.f;
    }

    // initializes spectral envelope leaky integrator constants
    b0 = RHO;
    a1 = 1 - RHO;

}


OLA::~OLA(){
    SAFE_DELETE( window );
    SAFE_DELETE( previous_half );
    SAFE_DELETE( ola_buffer );
    SAFE_DELETE( current_block );
    SAFE_DELETE( spectral_envelope );
    SAFE_DELETE( last_envelope );
}




void OLA::processBlock( float * block, unsigned int num_frames, float * output  ) {

```

```
    assert( M == num_frames );

    int i;

    // using the previous half plus the first half of the new input
    for (i = 0; i < R; i++) current_block[i] = previous_half[i];
    // the input array has 2 channels, but we are handling monophonic OLA
    for (i = R; i < M; i++) current_block[i] = block[2*(i-R)];
    this->processOverlappedFrame( current_block, &output[0] );


    // using exclusively the new block data
    // the input array has 2 channels, but we are handling monophonic OLA
    for (i = 0; i < M; i++) current_block[i] = block[2*i];
    this->processOverlappedFrame( current_block, &output[R] );


    // store the previous half, to use it with the next input block
    for (i = R; i < M; i++) previous_half[i-R] = block[2*i];

}


void OLA::processOverlappedFrame( float * input, float * output ) {

    // go to the freq. domain, perform spectral modifications and come back to
    // time domain
    this->spectralProcess( input );

    int i;

    // update the ola_buffer
    for (i = 0; i < N; i++)
        ola_buffer[(i+ola_buffer_head) % N] += input[i];

    for (i = 0; i < R; i++) {
        // we have all the overlaps needed for the "first" R samples in the
        // ola_buffer, so we output them
        output[i] = ola_buffer[(i+ola_buffer_head) % N];
        // zero-out the "first" R samples in the ola_buffer, to be ready for the
        // next iteration
        ola_buffer[(i+ola_buffer_head) % N] = 0.f;
    }
    // update the ola_buffer read head
    ola_buffer_head = (ola_buffer_head + R) % N;
}


void OLA::spectralProcess( float * input ) {
    // window the input to the first M samples
    MoFFT::apply_window( input, window, M );

    int i;
```

```

    // zero-padding
    for (i = M; i < N; i++) input[i] = 0.f;

    // forward FFT
    MoFFT::rfft( input, N, true );


    // Smooth out the spectral envelope using a Leaky Integrator
    for (i = 0; i < Nover2; i++)
        last_envelope[i] = b0*spectral_envelope[i] + a1*last_envelope[i];


    // Apply spectral processing
    for (i = 0; i < Nover2; i++) {
        // TODO: spectral processing in dB space
        // spectral processin in dB space
        // quick & dirty hack: squaring the filter values (they are between 0-1)
        // to make the differences more dramatic.
        input[2*i] *= last_envelope[i]*last_envelope[i];
        input[2*i+1] *= last_envelope[i]*last_envelope[i];
    }

    // inverse FFT
    MoFFT::rfft( input, N, false );

}


void OLA::updateSpectralEnvelope( float * new_envelope, unsigned int num_points ) {

    if (num_points == 0) return;

    // This implementation accepts input envelopes of a different size (not
    // necessarily Nover2). If the size doesn't match, inpterpolation is
    // performed. For now, a simple linear interpolation is being used. If I
    // have time I'll try to implement a more elegant interpolation
    int i;
    if ( num_points == Nover2 ) {
        for (i = 0; i < Nover2; i++)
            spectral_envelope[i] = new_envelope[i];
    } else {
        float step = ( (float)num_points - 1) / ( (float)Nover2 - 1 );
        double int_part;
        double frac_part;
        double idx = 0.f;
        i = 0;
        while (idx <= num_points) {
            frac_part = modf( idx , &int_part );
            if ( frac_part == 0) {
                spectral_envelope[i] = new_envelope[(int)int_part];
            } else {
                spectral_envelope[i] = lerp( new_envelope[(int)int_part],
```

```
                                                    new_envelope[(int)(int_part + 1)],
                                                    frac_part );
            }

            if ( spectral_envelope[i] > 1.f) spectral_envelope[i] = 1.f;
            else if ( spectral_envelope[i] < 0.f) spectral_envelope[i] = 0.f;

            idx += step;
            i++;
        }
    }

}



#endif // end __OLA_MM__
```