Audio Programming 4
Audio Visualiser

0800706M

May 4, 2012

Abstract

This paper presents a novel method of visualising digital audio signals. It is intended be aesthetically engaging, and also display a faithful representation of the original audio. The images created offer a sense of the spectral and temporal properties of the audio, with a general emphasis being artistic rather than analytic. Additionally, basic keyboard interaction is introduced to allow the user to tailor the visual output based on the particular audio input and personal taste.

1 Concept

The basis of the visualiser is the Hilbert transform. This is a linear operator which, when applied to a given real function $f(t)$, returns a function $\hat{f}(t)$ which has a $\frac{\pi}{2}$ radian phase-shift and is orthogonal to the original function. Mathematically it is defined as:

$$\hat{f}(t) = \frac{1}{\pi} P \int_{-\infty}^{\infty} \frac{f(\tau)}{t-\tau} d\tau \quad \text{if the integral exists} \tag{1}$$

where P is the Cauchy principal value. The mathematical proof and implementation are beyond the scope of this report, but are discussed in detail by Mathias Johansson (1999)[1]. The Python scipy library provides a function to calculate the Hilbert transform of a number array, which returns a set of complex numbers with the real parts representing the original signal, and the imaginary parts representing the Hilbert transform.

The Hilbert transform is more commonly used to transform a real signal into a so-called "strong analytic signal". Using the fact that $f(t)$ and $\hat{f}(t)$ are orthogonal, standard complex number arithmetic can be used to give the instantaneous amplitude and phase. This in turn can be thought of as the sum of many rotating vectors, each representing a single frequency component. Hence, plotting this instantaneous vector on an argand diagram as it rotates in time generates interesting visual patterns that directly represent the original audio signal.

The following basic steps are the algorithm for generating each frame of video:

1. Get audio buffer from sound card.
2. Calculate colour to display based on RMS level of audio.
3. Perform Hilbert transform on audio vector.
4. Scale and shift to fit onto window.
5. Filter to remove sharp corners and other artifacts.
6. Convert to list of (x, y) co-ordinates for plotting.
7. Draw black rectangle with high transparency.
8. Draw line joining all co-ordinates.
9. Draw text if needed.

2 Python Implementation

In the following sections, numbers in square parentheses refer to lines of the code found in appendix B.

2.1 The World

In order to draw images in a window, the Pyglet library was used[2]. This is essentially a wrapper around the OpenGL drawing functions[3], and also includes ways to schedule updates, control the frame rate, and other useful functions.

The basis of the entire application is the Window class in the pyglet.window library. This class has several special functions that it understands, but still must be programmed before anything is displayed. Hence, Window is used as the base of the World class. This solves many problems associated with defining the special Window functions, as the World class inherits these functions and they can be defined within World. It also allows for effective time keeping, as the 'dt' [108] value denoting the time since the last update can be accumulated as an attribute of the World, rather than using a global variable.

On initialisation of the World, the Window's __init__() function is called in order to generate the actual window [26]. Following this, the window is cleared, OpenGL and scheduler settings are given, the needed variables and objects are declared, and the audio stream is opened.

2.2 Real–time audio

Raw audio data is obtained using two separate modules. Firstly, the PyAudio python module is used to open an audio stream and return raw data[4]. Secondly, the Soundflower application generates a virtual sound card that PyAudio can read from[5]. The sound output of the computer is also routed to Soundflower so that any sound heard on the speakers is also sent to the visualiser application.

A PyAudio stream is opened with minimal options [51–56], as most of the default settings are adequate for this application. The options used are: 16-bit integer sample

values, 1 channel, 44.1kHz sample rate, input active, output inactive, and 1024 samples per audio frame. The audio vector size was chosen as a trade-off between fast reaction-time and preventing audio buffer overflow. Only one channel is used because a typical audio recording played into the visualser will not have particularly disparate stereo channels. For reasons discussed in section 2.3.1 a better response is obtained from lower frequencies, which tend to be panned towards the centre. These factors, in conjunction with the high processing needs of the application, influenced the decision to ignore the other stereo channel. All other values are standard.

A function get_audio() [80–90] is defined to retrieve audio samples from the stream. The data obtained from PyAudio is formatted by default as a single string with consecutive pairs of bytes representing a 16-bit audio sample. Hence the numpy.frombuffer() and numpy.array() functions are used to convert the string into individual sample values.

Due to the time between calls to get_audio() not being strictly defined, (it depends on the render time of the video), some error handling must be introduced to ensure that the application does not throw an exception during buffer overruns. If an exception occurs, the error message is checked to see if it was due to an overrun, and either raised or passed depending on the result. If the exception was due to an overrun, it is assumed that all audio samples are zero.

2.3 Audio processing

As previously mentioned, the basis of the DSP chain is the scipy.signal.hilbert() function. In addition to this, several custom classes and functions were created to perform some basic tasks.

2.3.1 Slide filter

The slide filter smooths signal logarithmically between changes in sample values[6]. The equation is given by:

$$y[n] = y[n-1] + \frac{x[n] - y[n-1]}{a} \quad \text{for } a \geq 1 \tag{2}$$

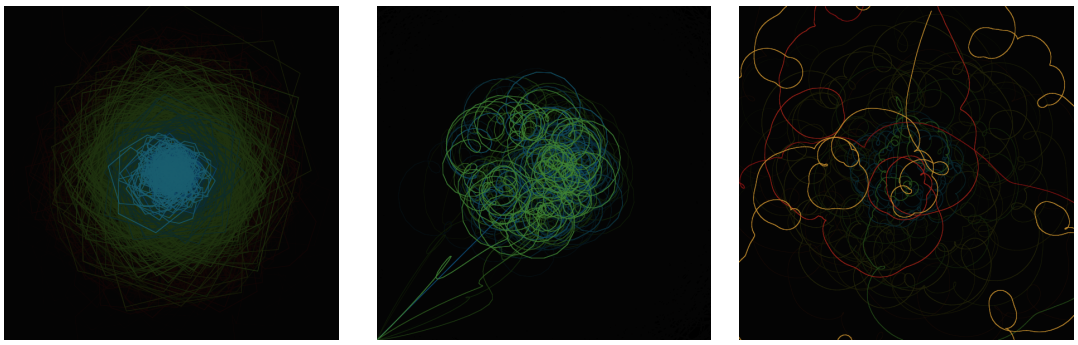
where x is the input, y is the output, and a is the slide value.

It was decided to use this filter over one from a pre-existing library for several reasons. Firstly, a custom class retains sample values from the previous audio vector, which (assuming a buffer overrun does not occur), minimises the distance between the beginning of a line and the end of the previous one. Scipy functions generally created undesired transient effects at the beginning of the lines.

Secondly, this filter is extremely simple, and does not require the processing power associated with FFT filters or their equivalent time–domain convolution operations. Lastly, it looked better on–screen than a normal one–pole or biquad low–pass filter, and as it did not need to filter to any frequency specification, the 'look' was deemed the most important factor.

The Silde_filter class itself comprises of a 2-sample buffer to save the current and previous outputs. It has also been designed to perform the filter operation on an entire audio vector in order to keep processing styles constant throughout all sections of the DSP chain.

This filter had the added bonus of segregating frequencies in the image. That is to say that because higher frequencies are damped, they tend as generate curves in the centre of the image, and lower frequencies spread the curves toward the edges. While this is not a strict rule, and was not true in all cases, the general effect proved aesthetically pleasing.



(a) Without filtering (b) Scipy filters (c) Slide filter

Figure 1: Screenshots showing typical unsuccessful (a) (b) and successful (c) filtering.

2.3.2 RMS

In order to introduce colour to the visualiser, the average power value was used to select a display colour. A function, rms() [172–174], was introduced to calculate the root–mean–square value of the audio buffer. The RMS was calculated thusly:

$$x_{rms} = \sqrt{\frac{1}{n}\sum_{i=0}^{n}(x_i)^2} \tag{3}$$

where x is a single audio sample. In practice, the audio buffer was also heavily scaled in order to reduce the range of the RMS to something close to $0 \leq \text{RMS} \leq 1$. The *exact* boundary values were not important.

2.4 Drawing

All drawing is performed by OpenGL commands, and is scheduled by the Pyglet clock. This works efficiently by transferring relatively short lists of vertices to video RAM, and commanding OpenGL to interpret and draw them using the graphics processor, thereby freeing the CPU and allowing it to handle all the other tasks.

2.4.1 Calculating vertices

The hilbert() function returns a list of complex numbers, which are subsequently filtered. As Python stores complex numbers in $a + jb$ format, they can already be interpreted as co-ordinages on an argand diagram. Vertices should generally fall between 0 and the width/height of the Window (measured in pixels), so the buffer is scaled using the World's 'self.scale' variable. It is also shifted by half the width and height of the window to bring the origin into the centre [99–101].

As an extra, the standard 'oscilloscope style' waveform can be viewed. The vertices for this are calculated from the absolute value of the post-filter audio sample, and its index within the buffer. Again, they are scaled to fit the window, and shifted to the centre [102-106]. The buffer is also downsampled by half, as this number of data points fitted conveniently on the window.

In both of these cases, the filtered buffer is unpacked into a tuple of floats, with each subsequent pair representing an x and y co-ordinate.

2.4.2 Drawing to the window

As with many audio visualisers, it was desirable to emulate the persistence–of–vision effects of old CRT type monitors. To this end, at the start of each frame, a black polygon with an opacity of 10% is drawn over the entire window [64–68]. Over many frames, this gives the impression that the audio trace is fading away.

Depending on the 'self.mode' variable, one or both of the vertex lists and their associated colour tuples are passed to the pyglet.graphics.draw() function [69–75]. This has the effect of drawing a 'GL primitive' to the window. There are several types of primitive, but in this case, a simple line joining all vertices is chosen. This is all that is required to display the audio waveform.

Text can also be displayed, depending on whether it is scheduled or not [76–78]. Since only one piece of text is to be displayed at one time, a variable 'self.last_text' is defined, which is compared to the current time. If the current time is less than one second after the last text update, the text is refreshed. If not, it is left to fade naturally.

2.5 Extra functionality

2.5.1 Colour interpolation

As mentioned in section 2.3.2, the RMS value of the audio is used to determine the colour of the line being drawn. This value is then used as an interpolating index into an array of RGB colour values. The colours were chosen purely subjectively, and then arranged as a list of tuples, with the 'quietest' one first [177–180].

Seeing as the there are four colours to interpolate between, each is given an equal sector of the range $0 \leq \text{index} \leq 1$. It is then assumed that the midpoint of a sector represents a pure colour with no interpolation. Given this, the interpolation index is calculated and the two colours to interpolate between are determined. The fractional part of the index is then used as a blend value, which determines how much of each

colour to include in the final value: $c_{out} = c_2 b + c_1(1 - b)$ where c is a colour, and b is the blend value.

Note: Since the Hilbert transform gives the instantaneous amplitude, it was attempted to give each point a different colour. In practice however, this proved too slow to render, which resulted in constant buffer overruns. This method was therefore rejected.

2.5.2 Keyboard input

It is inevitable that different audio files will have different sonic properties. This is the motivation behind giving the user some form of control over the display. The on_key_press() function inherited from the Window class is used to implement this [111–134]. The user has control of the scaling of the visualiser, as well as the amount of filtering performed on the audio.

As a small extra, the keyboard input also offers options to hold the visualiser steady; switch between the hilbert display, waveform display, or both; and to save the current frame as PNG image. A message is displayed on the screen when any of the key actions are invoked.

2.5.3 Screenshots

A Screenshot_saver class is defined in order to save a image [151–168]. The first task is to create a folder to hold the screenshots if one does not already exist. The Python os library has functions to do exactly this [154–156].

When the save_image() function is invoked, the current video buffer is obtained from the Pyglet Buffer_manager class. As OpenGL has been setup to use alpha blending, the alpha channel must be stripped from the screenshot, otherwise all black areas appear as transparent in the PNG image. This is done simply by setting the 'format' attribute of the Image_data object.

The Python datetime library is then used to construct a filename that contains the exact time and date. This prevents overwriting any current screenshots. Finally, the function returns a string depicting whether the file has been saved successfully or not. As with the get_audio() function, the exceptions must be handled correctly to prevent the application crashing if the file is not saved for any reason.

3 Conclusion

The application presents an alternative method of visualising pure audio data. The patterns vary greatly depending on the audio input used, such that while the visualiser is predictable, it remains appealing over prolonged use. The ability of the user to alter the behavior slightly in real-time, and save screenshots adds interactivity for the user.

Some improvements can be made with further development:

- The audio could be upsampled to give more co-ordinates, and hence smoother lines at higher frequencies. This could require 'thinning' of the vertex list to remove unnecessary co-ordinates.
- The application could play an audio file as opposed to simply taking input from a sound card.
- The parameters of the animation could vary over time, allowing an even more interesting visual experience.

References

[1] M. Johansson, "The Hilbert transform," Master's thesis, Växjö University, 1999.

[2] "Pyglet python library." `http://www.pyglet.org/`, January 2010.

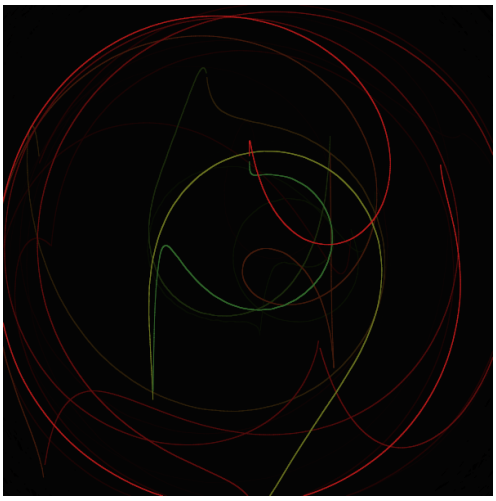
[3] Fallout Software, "Introduction to opengl primitives - drawing basic shapes." `http://www.falloutsoftware.com/tutorials/gl/gl2p5.htm`, November 2009.

[4] H. Pham, "Pyaudio python library." `http://people.csail.mit.edu/hubert/pyaudio/`, 2010.

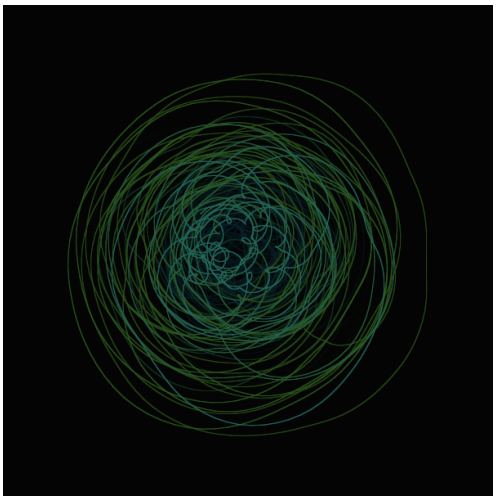
[5] Cycling 74, "Soundflower: Cycling 74." `http://cycling74.com/soundflower-landing-page/`, 2012.

[6] Cycling 74, "slide~ Reference." `http://www.cycling74.com/docs/max5/refpages/msp-ref/slide~.html`.

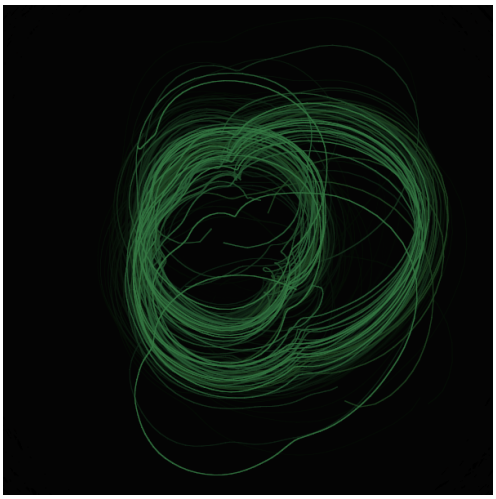
A Example screenshots



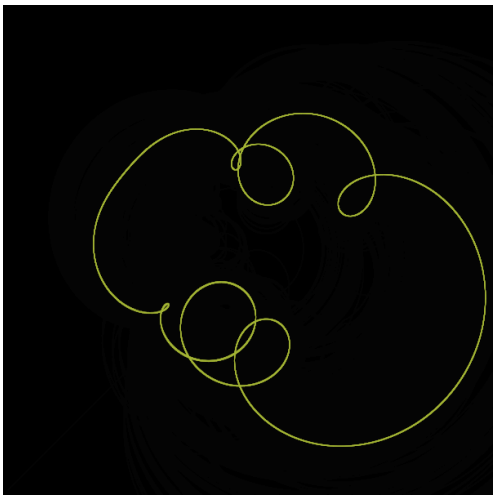
(a) James Blake – Limit to Your Love: Sub bass and minimal drums



(b) Debussy – Prelude a l'apres-midi d'un faune: Orchestral



(c) Flogging Molly – Grace Of God Go I: Solo male voice



(d) 86Hz oscillator with partials 1–8 at various levels

Figure 2: Screenshots taken from real–time audio streams

B Full code listing

```
import pyglet
from pyglet.gl import *
from pyglet import clock
from pyglet.window import key
from scipy.signal import hilbert
import numpy as np
import pyaudio
import math
import pylab
import cmath



### Parameters for creating the World ###
my_audio_params = ( pyaudio.paInt16, 1, 44100, True, False, 1024 )
WIDTH = 600
HEIGHT = 600
FPS = 100


### Class definitons ###
class World(pyglet.window.Window):
    def __init__(self, w, h, fps, audio_params):
        if w and h and fps > 0:
            pyglet.clock.schedule_interval(self.update, 1/float(fps))
            super(World, self).__init__(width=w, height=h)
            self.clear()
        else:
            raise Exception('Invalid width, height or fps')
        self.display_mode = 1    # 1: Just hilbert
                                 # 2: Just waveform
                                 # 3: Both
        pyglet.clock.set_fps_limit(fps)
        glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)
        glEnable(GL_BLEND)
        glEnable(GL_LINE_SMOOTH)
        glLineWidth(2)
        self.time = 0.
        self.last_text = None
        self.text = None
        self.audio_params = audio_params
        self.line_points = np.zeros(self.audio_params[5]*2)
        self.line_colour = (0.5, 0.5, 0.5, 0.8)
        self.wave_display = np.zeros(self.audio_params[5])
        self.scale = 0.04
        self.hold = False
        self.fade_time = 10.
        self.f = Slide_filter(50)
        self.ss = Screenshot_saver()
        self.p = pyaudio.PyAudio()
        self.stream = self.p.open(  format = self.audio_params[0],
```

```
                                    channels = self.audio_params[1],
                                    rate = self.audio_params[2],
                                    input = self.audio_params[3],
                                    output = self.audio_params[4],
                                    frames_per_buffer = self.audio_params
                                        [5])

    def put_text(self,string):
        self.text = string
        self.last_text = self.time

    def on_draw(self):
        if self.hold == False:
            pyglet.gl.glColor4f(0.,0.,0.,0.1)
            w = self.width
            h = self.height
            glLineWidth(2)
            pyglet.graphics.draw(4, pyglet.gl.GL_POLYGON,('v2i', (0,0,0,h
                ,w,h,w,0)))
            if self.display_mode in [1,3]:
                r,g,b = self.line_colour
                pyglet.gl.glColor4f(r,g,b,0.8)
                pyglet.graphics.draw(len(self.line_points)/2, pyglet.gl.
                    GL_LINE_STRIP, ('v2f', self.line_points))
            if self.display_mode in [2,3]:
                pyglet.gl.glColor4f(0.,1.,0.,0.8)
                pyglet.graphics.draw(len(self.wave_display)/2, pyglet.gl.
                    GL_LINE_STRIP, ('v2f', self.wave_display))
            if self.text:
                if self.time-self.last_text < 1:
                    pyglet.text.Label(self.text, bold=True, font_size=14,
                        color=(200,200,200,100), x=10, y=10).draw()

    def get_audio(self):
        try:
            raw = self.stream.read(self.audio_params[5])
        except IOError as ex:
            if ex[1] != pyaudio.paInputOverflowed:
                raise
            else:
                print "Warning: audio input buffer overflow"
                pass
            raw = '\x00' * self.audio_params[5]
        return np.array(np.frombuffer(raw,np.int16), dtype=np.float64)


    def update(self,dt):
        audio = self.get_audio()
        self.line_colour = interp_colour(rms(audio/10000.))
        shifted = hilbert(audio)*self.scale
        shifted = np.add(shifted, self.width/2+(self.height/2)*1j)
        shifted = self.f.filter(shifted)
        for i, co_ord in enumerate(shifted):
```

```
100             self.line_points[2*i] = co_ord.real
101             self.line_points[(2*i)+1] = co_ord.imag
102         wave_offset_y = self.height*0.5*(1-math.sqrt(2)) # To account for
                the 200+200j added earlier
103         wave_offset_x = (self.width - len(shifted)/2)/2
104         for i, co_ord in enumerate(shifted[::2]):
105             self.wave_display[2*i] = i + wave_offset_x
106             self.wave_display[(2*i)+1] = abs(co_ord) + wave_offset_y
107         self.on_draw()
108         self.time += dt
109         #print self.time
110 
111     def on_key_press(self,symbol,modifiers):
112         if symbol == key.Q:
113             my_world.f.a = max(1., my_world.f.a*1.2)
114             self.put_text("Filtering value: "+str(my_world.f.a))
115         elif symbol == key.A:
116             my_world.f.a = max(1., my_world.f.a/1.2)
117             self.put_text("Filtering value: "+str(my_world.f.a))
118         elif symbol == key.W:
119             my_world.scale += 0.002
120             self.put_text("Scale: "+str(my_world.scale))
121         elif symbol == key.S:
122             my_world.scale -= 0.002
123             self.put_text("Scale: "+str(my_world.scale))
124         if symbol == key.SPACE:
125             my_world.hold = not my_world.hold
126         if symbol == key.P:
127             self.put_text(self.ss.save_image())
128             self.last_text = self.time+3 # Account for the time taken to
                    save the file
129             self.hold = False
130         if symbol == key.Z:
131             if self.display_mode < 3:
132                 self.display_mode += 1
133             else:
134                 self.display_mode = 1
135 
136 
137 class Slide_filter():
138     def __init__(self, a):
139         self.buffer = np.zeros(2, dtype='complex128')
140         self.a = a
141 
142     def filter(self, input_buff):
143         b = np.zeros(len(input_buff), dtype='complex128')
144         for i in range(len(input_buff)):
145             self.buffer[1] = self.buffer[0]
146             self.buffer[0] = self.buffer[1] + ((input_buff[i]-self.buffer
                    [1]) / self.a)
147             b[i] = self.buffer[0]
148         return b
149 
```

```
150
151 class Screenshot_saver():
152     def __init__(self):
153         self.m = pyglet.image.get_buffer_manager()
154         import os
155         if not os.path.exists('screenshots'):
156             os.mkdir('screenshots')
157
158     def save_image(self):
159         from datetime import datetime
160         img = self.m.get_color_buffer().get_image_data()
161         img.format = 'RGB'
162         d = datetime.now().strftime("_%I.%M.%S_%d-%m-%Y")
163         filepath = 'screenshots/image'+d+'.png'
164         try:
165             img.save(filepath)
166             return str(filepath)+" saved"
167         except:
168             return "Screenshot could not be saved"
169
170
171 ### Miscellaneous function definitions ###
172 def rms(buff):
173     rms_val = math.sqrt(sum(np.multiply(buff,buff))/len(buff))
174     return rms_val
175
176 def interp_colour(pos):
177     c = [(14, 82 , 127), # blue
178         ( 69, 138, 44 ), # green
179         (208, 203, 57 ), # yellow
180         (196, 28 , 28)]  # red
181     f = np.divide(c,255.)
182     sector1 = max(min((pos*len(f))-0.5, float(len(f)-1)), 0.)
183     sector2 = max(min((pos*len(f))+0.5, float(len(f)-1)), 0.)
184     f1 = f[int(sector1)]
185     f2 = f[int(sector2)]
186     blend = sector1 - int(sector1)
187     return (f2*blend) + (f1*(1.-blend))
188
189
190
191 ### --- MAIN CODE --- ###
192 my_world = World(WIDTH, HEIGHT, FPS, my_audio_params)
193 pyglet.app.run()
```