Pulse Finder

/ OOP Project

BONHEUR SAVINIEN - BSCV

SUMMARY

Introduction:

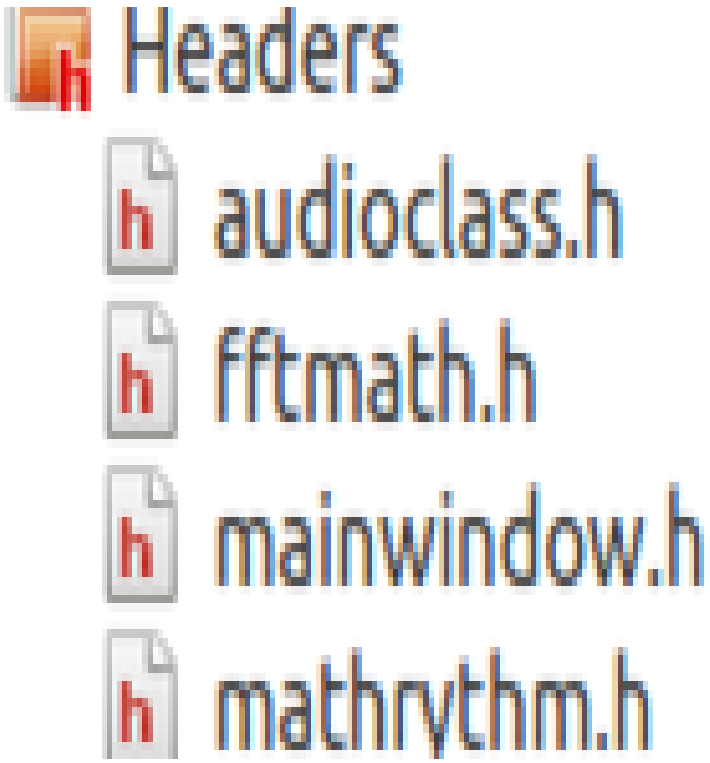
The following page describe my project, the rhythm finder. As the code is all commented I will not explain everything in details but try to give a quick overview of the interface, the general concept, the idea driving the code and is construction.

I. QUICK OVERVIEW OF THE CLASS AND OF THE UNDERLYING WORKING THROUGH THE INTERFACE

a) Class

The program is made of 4 independents classes (no inheritance):

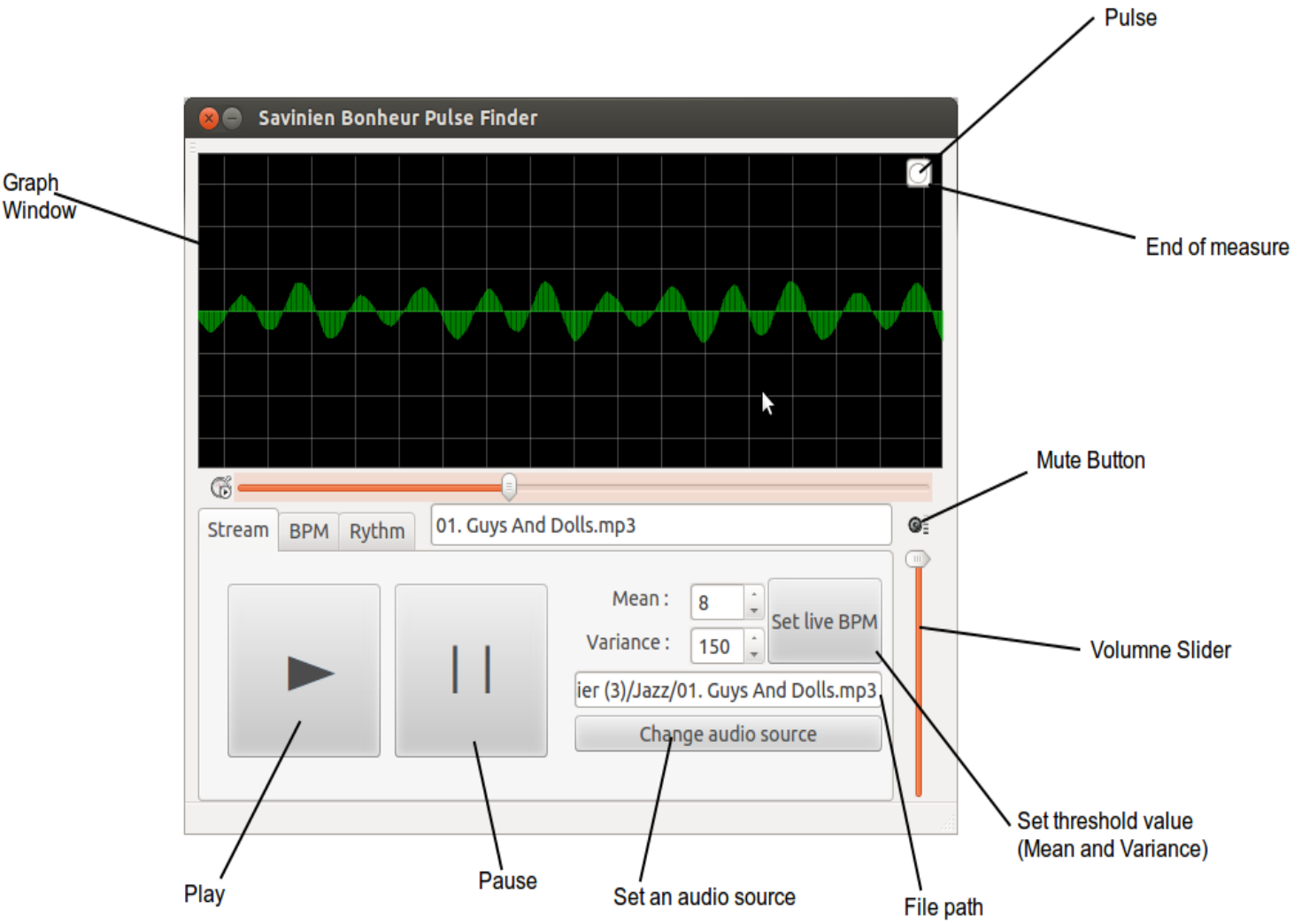
- The Audioclass Class: Is here to deal with getting and sending audio data, also checking audio file validity
- The FFT Class: provide an Fast Fourier transform radix 2 with Hamming windowing and Derivation
- The Main window Class: provide an interface for the user, also provide a metronome
- The mathrythm Class: Analyze the data to extract pulse and bpm.



b) Interface

This part is the users access to the program.

Although it's pretty much explanatory I will describe globally the action of each button:



- Play button

Emit a signal from main window to play the audio media.

- Pause button

Emit a signal from main window to pause the audio media.

- File path

Line edit to input a file path. Change momently is color accordingly to the validity of the path.

- Set an audio source

Grab the file path stored in line edit, check is validity, emits a signal toward line edit to communicate this validity, and if valid change the audio path.

- Volume slider

Change the volume.

- Seek slider

Allow the user to move inside the media.

- End of measure

When the metronome function is used this part change is color to orange at the end of each measure (blank otherwise).

- Pulse

A measure is divide in time, this button is checked at each time.

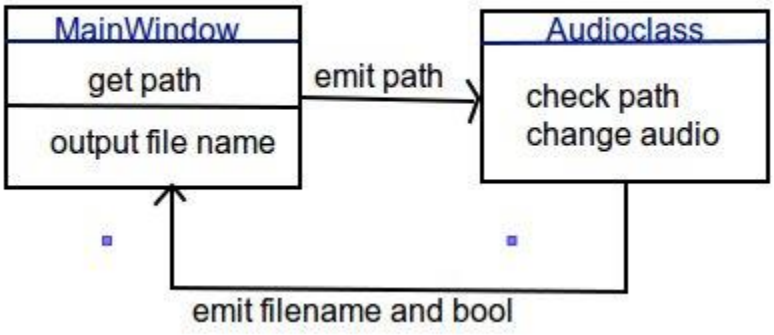
- Graph window

Display the sound graph either in time domain (draw the last 1024 intensity value send to the speaker) or in Frequency domain via a FFT transformation. In Frequency domain the scales are Hertz for the abscise and x^10 for the ordinate (without unit name).

II. AUDIO AND VISUAL OUTPUT

a) Change path functionality

When you set a new path, the main window gather the data from the path line edit and send them via a signal to the Audio Class. The Audioclass then check the validity of it and send back the validity of the path and the filename.



The code is pretty straightforward and self-explanatory so I will only explain how we check the path:

```
void Audioclass::changepath(const QString &path){

    //create a bool to check if the path is good
    bool ispathgood=false;

    //Create a source object to use is possibility of path checking
    Phonon::MediaSource source(path);

    //Check if the file path is not already used (if so no need to change it)
    if  path!=get_currentpath()                    {

        //Check the path type (-1 is invalid and 0 is path that we can use)
        if (source.type()==0){

            ispathgood=true;

            //change the file path (the music is automaticaly stopped
            loadfiletomedia->setCurrentSource(path);
```

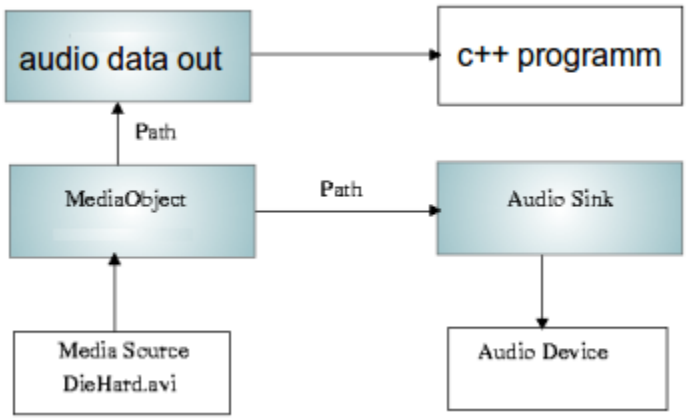
Here we first check if the path is not already the one set (to avoid a music reset if the user click again on set path) with the first *if*.

The second *if* is the one we check if path is valid. To do so we use a function of the media source object, this function return us the path type (see below), if you use an audio file the type will be 0 if it's anything else (from a cd to a gibberish input) the source type will not be equal to 0.

Constant	Value	Description
Phonon::MediaSource::Invalid	-1	The MediaSource object does not describe any valid source.
Phonon::MediaSource::LocalFile	0	The MediaSource object describes a local file.
Phonon::MediaSource::Url	1	The MediaSource object describes an URL, which can be either a local file or a file on the network.
Phonon::MediaSource::Disc	2	The MediaSource object describes a disc, e.g., a CD.
Phonon::MediaSource::Stream	3	The MediaSource object describes a data stream. This is the type used for QIODevices. Note that a stream opened with a QUrl, will still be of the Url type.
Phonon::MediaSource::Empty	4	The media source doesn't have a source.

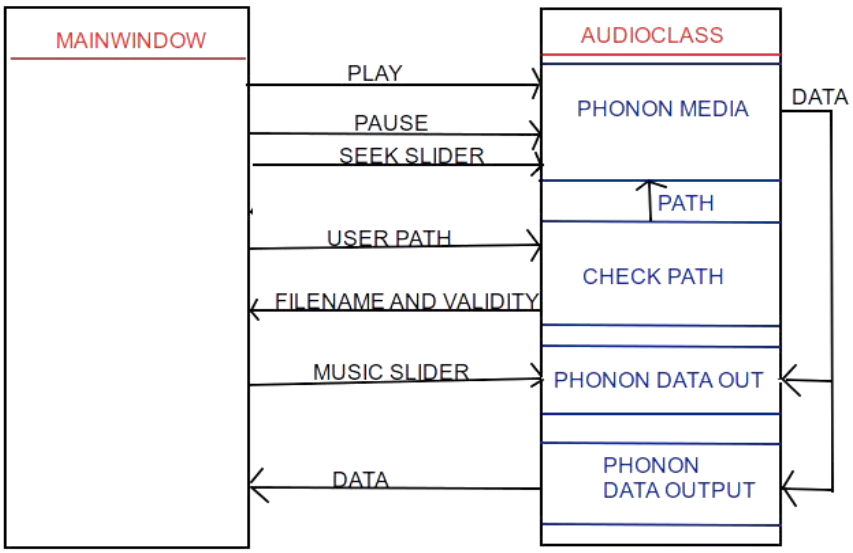
b) The playback sound functionality

First off all we need to explain how the Audioclass link the C++ code, the mediafile (mp3, wave ...) and the speaker:



As we can see in the above diagram (partially taken from : QT4/doc) we create a media object which gather data from the audio file and them send them via two buffers to respectively the speaker (audio device) and the program itself.

When you first click (we use a Boolean to only create and connect once the audio object) on the set path we will create the following connection:



Remark: the media object (get sound data) and the audio out object (send sound data to speaker) are passed to the audio class from the main window to use respectively the seek slider and the volume slider.

As all the path are set we then communicate with the audio class only by signal and slot.

c) The display function

This part will be cut into two sections, the frequency and the time domain:

Time domain

For the time domain we simply take all the data from the left channel and the right channel index by index, and for each index add them and display them along the x axis while taking care of the size of the window.

```
for(int i =0; i < nSamples ; i++)//for all the value showable do
{
    y = ((m_audiodata[Phonon::AudioDataOutput::LeftChannel][i])+(m_audiodata[Phonon::AudioDataOutput::LeftChannel][i]))/2;
    y=y/2*h/51070 + 205-b*29;//set the height of the graph
    x = (i*w)/nSamples+10;//set the distance between two value (dx)
    painter.drawLine(x,205-b*29,x,y);
}
```

Frequency domain:

For the frequency domain we pass the data from the buffer of 1024 data (the same that we use for the time domain display) through an FFT (we create a vector of complex number with the Left audio channel as the real part and the Right audio channel as the imaginary part of the signal to pass both channel in one calculation).

We take half of the signal (as an FFT is mirrored by is center) separate into 16 bin (512/32=16, each bin represent a mean of the values that compound it). The value that compound the bin are the log of the magnitude at each index (the magnitude to pass the signal from complex to real and the log to fit everything into the black window)

```
for(int i =0; i < nSamples/2 ; i++)//as the FFT is symetrics by is middle
{
    complex<float> value;
    value=audiodata.at(i);

    numberofvalues=numberofvalues+1;

    if (numberofvalues==32)//every 32 value added draw is value
    {
        subbandnumber=subbandnumber+1;
        bin=bin+20*log10(sqrt(pow( value.real(),2) +pow( value.imag(),2) ))
        bin=bin/32;

        x = subbandnumber*34+3;
        y= (bin*h)/10;
```

d) Metronome

The first step of a metronome is to have a clock, we use:

```
QTimer *timer = new QTimer(this);//Declare a Timer
  connect(timer, SIGNAL(timeout()), this,SLOT(clock())); //Connect the timer
  timer->start(10); //state the frequency in ms
```

wich call the call the slot *clock()* each 10 millisecond (if you go under you're not sure to be compatible with everyone's pc) this clock has a counter :

```
void MainWindow::clock(){//this function is called every 10 ms
  set_Counttime(get_Counttime()+10);//this edit the clock counter
  tempo();
}
```

Which is incremented every 10 Ms.

Afterward the tempo function just check if the clock is equal to 60000/get_Bpm () we know that we have a pulse. Another counter turn to find the end of the measure (if the measure is made out of 5 time the end is every 5 pulse).

III. MATH AND PULSE

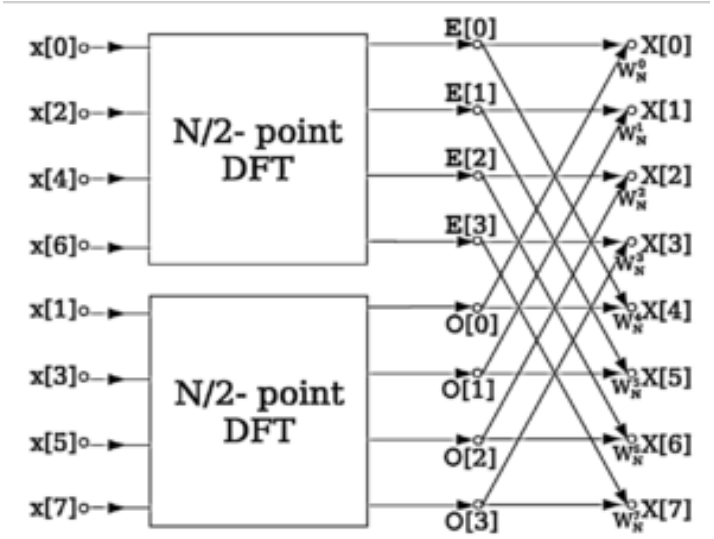
a) The FFT

How work an FFT: We know that when dealing with an array of discrete number we can describe their Fourier transformation as a divide and conquer algorithm where at each step (or at least while the size of the array is superior at 1) we apply the following operation (where k is the index):

$$\begin{aligned} X_k &= E_k + e^{-\frac{2\pi i}{N}k} O_k \\ X_{k+\frac{N}{2}} &= E_k - e^{-\frac{2\pi i}{N}k} O_k \end{aligned}$$

Remark: the exponential is called the twiddle factor and can be precalculated as it only depend on the size of the array and not on what it contains. Recalculating the twiddle factor give a great computation gain.

Which give us the following diagram (here for a size 8 array):



And the following pseudocode:

```
X_{0,...,N-1} ← ditfft2(x, N, s):
    if N = 1 then
        X_0 ← x_0
    else
        X_{0,...,N/2-1} ← ditfft2(x, N/2, 2s)
        X_{N/2,...,N-1} ← ditfft2(x+s, N/2, 2s)
        for k = 0 to N/2-1
            t ← X_k
            X_k ← t + exp(-2πi k/N) X_{k+N/2}
            X_{k+N/2} ← t - exp(-2πi k/N) X_{k+N/2}
        endfor
    endif
```

```
DFT of (x_0, x_s, x_2s, ..., x_(N-1)s):

        trivial size-1 DFT base case

          DFT of (x_0, x_2s, x_4s, ...)
          DFT of (x_s, x_s+2s, x_s+4s, ...)
        combine DFTs of two halves into full DFT:
```

(See https://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey_FFT_algorithm for more precision)

This explanation will be in three part:

1. The call function:

We dispose of different call function but will only talk of the one taking a map as input function and using a hamming window and a derivative (the other are working roughly the same but in simplest way).

First of all, we call the function by passing the size of the array, the map containing the data and a vector on which we will write.

Once the function called, we first check if the size is a power of 2 as the FFT is based on a radix 2. For that we do:

```
int a=log2(size);
float b=log2(size);
//check if the size is under the form (2^n) and return false if not
if (a!=b){
    cout<<"the FFT size must be under the form 2^n"<<endl;
    return false;
}
```

Here the idea is to use the fact that log2 (2^n) =n. This mean that for a power of 2 (and only for those) a log operation will output an int (which doesn't take into account any float) equal to a float (we keep float number).

If the condition is satisfied we will calculate all the twiddle factor:

```
//Calculate all the value for the twiddle exponantial array
for ( int i=0 ; i < (get_Size()/2); i++)
{
    temporarynumber.real( cos( pi * (-2) * (i) / get_Size() ));
    temporarynumber.imag( sin( pi * (-2) * (i) / get_Size() ));
    set_Twiddlearrayvalue( i , temporarynumber );
}
```

After that we fill the output vector with the derivate value which went through a Hamming windowing:

- Derivation (this step will accentuate sound transition)

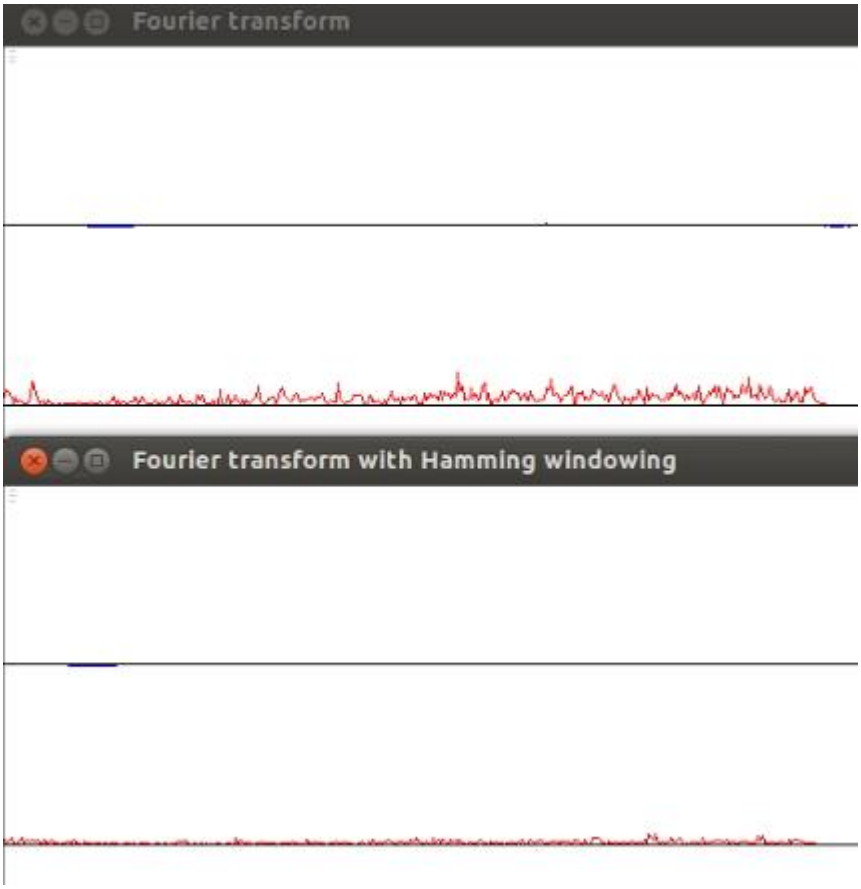
```
complex<float>  temporarynumber2;
temporarynumber.real( m_audiodata[Phonon::AudioDataOutput::LeftChannel][i+1]);
temporarynumber.imag(  m_audiodata[Phonon::AudioDataOutput::RightChannel][i+1]);



temporarynumber.real( m_audiodata[Phonon::AudioDataOutput::LeftChannel][i]);
temporarynumber.imag(  m_audiodata[Phonon::AudioDataOutput::RightChannel][i]);


temporarynumber.real(temporarynumber2.real()- temporarynumber.real());

temporarynumber.imag(temporarynumber2.imag()-temporarynumber.imag());
```

- Hamming filtering

```
//Here we apply a Hamming windowing to our data to get a better result as the FFT output
hammingwindowing.real(0.53836-(0.46164*cos( 2 * pi * i /get_Size() ))*(temporarynumber.real()));
hammingwindowing.imag(0.53836-(0.46164*cos( 2 * pi * i /get_Size() ))*(temporarynumber.imag()));
```

The hamming filtering is here to reduce the frequency signal distortion:

Finally the call function start the FFT.



2. The FFT function

Basically this function follow the following pseudocode:

```
X_0,...,N-1 ← ditfft2(x, N, s):
    if N = 1 then
        X_0 ← x_0
    else
        X_0,...,N/2-1 ← ditfft2(x, N/2, 2s)
        X_N/2,...,N-1 ← ditfft2(x+s, N/2, 2s)
        for k = 0 to N/2-1
            t ← X_k
            X_k ← t + exp(-2πi k/N) X_k+N/2
            X_k+N/2 ← t - exp(-2πi k/N) X_k+N/2
        endfor
    endif
```

```
DFT of (x_0, x_s, x_2s, ..., x_(N-1)s):

        trivial size-1 DFT base case

        DFT of (x_0, x_2s, x_4s, ...)
        DFT of (x_s, x_s+2s, x_s+4s, ...)
        combine DFTs of two halves into full DFT:
```

Nonetheless we have a slight difference on the twiddle factor (the exponential part) as we used precomputed twiddle factor. To do so we:

```
complex<float> comp= get_Twiddlearrayvalue(g)  *  (get_FFTvectorvalue(k+N/2));

g=g+(get_Size())/(N); //We dont navigate the same is the precalculate twiddle
```

The strange g index incrementation is due to the following constation:

We have recalculated Exp (-2*pi*k/N) where N is the size of the array and k the index. We can clearly see that the important part is k/N.

So if we have an array size 8 we get for k/N: 1/8 1/4 3/8 1/2 5/8 3/4 7/8 1
With k: 1 2 3 4 5 6 7 8

And for an array size 4 we get for k/N: 1/4 1/2 3/4 1
With k: 1 2 3 4

If we take try to find a logical progression to get the value of an array of size 4 within the array of size 8, we find k*2. This 2 come directly from 8/4. If we extend the concept we find an index progression as index=index+ (array size for which we build the twiddle array/ current array size).

3. The Cut function:

In this function we divide and conquer on the initial array in the purpose of avoiding a huge quantity of temporary array :

```
for (int k = 0+(n*(N/2)); k < (N+n*N/2) ; k++)
{
    temp[j]=get_FFTvectorvalue(k);//copy the a part of the array to divide and conquer on it
    j=j+1;
}
```

We copy the part of the array to sort in a temporary array:

N represent the first index from which we start to copy and N represent how much index we copy

Sort it:

```
for (int k=(0); k<=(N-1)/2; k++)
{
    copy[k]=temp[k*2+1];//copy the odd index out of the array
    temp[k]=temp[2*k];//move the even index in the second half of the array
}
for (int k=1; k<=((N-1)/2+1); k++)
{
    temp[k+(N-1)/2]=copy[k-1];//Copy the odd array back into the array in is first half
}

j=0;//set j at 0 to use it as an index again
```

Insert it back sorted:

```
for (int k=0+(n*(N/2)); k<(N+n*N/2); k++)
{
    set_FFTvectorvalue(k,temp[j]);//copy back
    j=j+1;
```

b) The live pulse finder

To more precision go to about the algorithm go to:
http://archive.gamedev.net/archive/reference/programming/features/beatdetection/

To find a pulse in live we suppose that a pulse is represented by a change in the variance and in energy compared to local value.
To use this hypothesis we first derive our signal (to accentuate transition), apply a Hamming windowing and then pass the signal through an FFT to pass it into the frequency domain.

We can remark that for our 1024 value of song sampled at 44100 Hz we will extract from the FFT signal going from 0 to 22050 hertz. Or the piano note with the highest frequency is at 4186.01 Hz, while the

violin is approximately at 10 000 Hz, this show us that the most important part of music will be under 10000 Hz and so we should create a way to accentuate that. That the point of the subbandlogdivision function.

The subband log division function:

This function take into account what the previous conclusion by applying a kind of logarithm frequency regrouping (the highest the frequency, the bigger the regrouping).

To do so we take into account that, as the FFT only work with 2^n value, we can divide it in subband of size 2^n. Here is a property 2^n=2^(n-1)+2^(n-1) and 2^(n-1)=2^(n-2)+2^(n-2), from that we applied the following regrouping :

From 0 to size/4: value= $\sum_{i=1}^{n} i$ where ”i“ is the frequency value at the index I and n<=2^2

From at size/4: value= $\sum_{i=1}^{n} i$ where “i” is the frequency value at the index I and n<=2^3

From size/4 to size/2: $\sum_{i=1}^{n} i$ value= where “i” is the frequency value at the index i and n<=2^k (where k start a 3 and increment itself (+1) at each iteration)

We store those calculated value into a vector passed by the pulse function with a pointer.

Historic and thresholding:

Now that we have cut our signal into subband we have to stock them (we will keep 127 of them) to stock them we create a map of vector. This map of vector present two interest over a vector of vector:

- It's easier to create and manipulate
- We can add a vector easily with the swap function, this function is already implemented and allow us to edit the historic by swapping pointer rather than by swapping all the value. Swapping pointer is faster and doesn't invoke vector size problems.

At each new value added we calculated the mean and the variance again with simple iterative loop. We then apply a simple thresholding on the mean and on the variance to check if there is a pulse and send a bool containing the result.

Conclusion:

All the objective weren't fulfilled but globally the project seems to be a success as I'm able to play audio, draw it and extract information from it.

Although my code is not perfect (far for it) I tried my best and learned a lot working on it. Some default subsist, some amelioration could be made (for example the FFT to draw the data on the graph is always running or I could have create a specific widget to contain the graph area), some security added but it served it purpose teaching me C++ and OOP.

Sources

- https://en.wikipedia.org/wiki/Main_Page
- http://doc.qt.io/qt-4.8
- http://stackoverflow.com/
- http://archive.gamedev.net/archive/reference/programming/features/beatdetection/