« Playing around with HTML 5

Music Matching (part deux) »

Written by Roy van Rijn (royvanrijn.com) on Jun 1, 2010 01:43:08 : 0 COMMENTS

Creating Shazam in Java

A couple of days ago I encountered this article: How Shazam Works

This got me interested in how a program like Shazam works… And more importantly, how hard is it to program something similar in Java?

About Shazam

Shazam is an application which you can use to analyse/match music. When you install it on your phone, and hold the microphone to some music for about 20 to 30 seconds, it will tell you which song it is.

When I first used it it gave me a magical feeling. “How did it do that!?”. And even today, after using it a lot, it still has a bit of magical feel to it.
Wouldn’t it be great if we can program something of our own that gives that same feeling? That was my goal for the past weekend.

Listen up..!

First things first, get the music sample to analyse we first need to listen to the microphone in our Java application…! This is something I hadn’t done yet in Java, so I had no idea how hard this was going to be.

But it turned out it was very easy:

```
final AudioFormat format = getFormat(); //Fill AudioFormat with the wanted settings
DataLine.Info info = new DataLine.Info(TargetDataLine.class, format);
final TargetDataLine line = (TargetDataLine) AudioSystem.getLine(info);
line.open(format);
line.start();
```

Now we can read the data from the TargetDataLine just like a normal InputStream:

```
// In another thread I start:

OutputStream out = new ByteArrayOutputStream();
running = true;

try {
   while (running) {
      int count = line.read(buffer, 0, buffer.length);
      if (count > 0) {
```

```
            out.write(buffer, 0, count);
        }
    }
    out.close();
} catch (IOException e) {
    System.err.println("I/O problems: " + e);
    System.exit(-1);
}
```

Using this method it is easy to open the microphone and record all the sounds! The AudioFormat I'm currently using is:

```
private AudioFormat getFormat() {
    float sampleRate = 44100;
    int sampleSizeInBits = 8;
    int channels = 1; //mono
    boolean signed = true;
    boolean bigEndian = true;
    return new AudioFormat(sampleRate, sampleSizeInBits, channels, signed, bigEndian);
}
```

So, now we have the recorded data in a ByteArrayOutputStream, great! Step 1 complete.

Microphone data

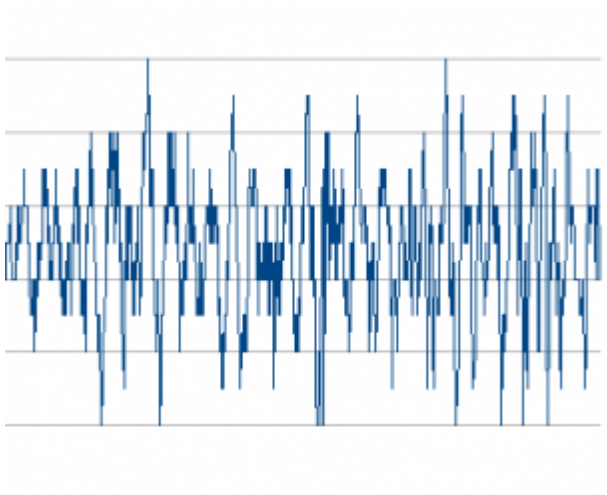
The next challenge is analyzing the data, when I outputted the data I received in my byte array I got a long list of numbers, like this:

```
0
0
1
2
4
```

```
7
6
3
-1
-2
-4
-2
-5
-7
-8
(etc)
```

Erhm… yes? This is sound?

To see if the data could be visualized I took the output and placed it in Open Office to generate a line graph:



Ah yes! This kind of looks like ‘sound’. It looks like what you see when using for example Windows Sound Recorder.

This data is actually known as time domain. But these numbers are currently basically useless to us… if you read the above article on how Shazam works you’ll read that they use a spectrum analysis instead of direct time domain data.
So the next big question is: How do we transform the current data into a spectrum analysis?

Discrete Fourier transform

To turn our data into usable data we need to apply the so called Discrete Fourier Transformation. This turns the data from time domain into frequency domain.
There is just one problem, if you transform the data into the frequency domain you loose every bit of information regarding time. So you'll know what the magnitude of all the frequencies are, but you have no idea when they appear.

To solve this we need a sliding window. We take chunks of data (in my case 4096 bytes of data) and transform just this bit of information. Then we know the magnitude of all frequencies that occur during just these 4096 bytes.

Implementing this

Instead of worrying about the Fourier Transformation I googled a bit and found code for the so called FFT (Fast Fourier Transformation). I'm calling this code with the chunks:

```
byte audio[] = out.toByteArray();

final int totalSize = audio.length;

int amountPossible = totalSize/Harvester.CHUNK_SIZE;

//When turning into frequency domain we'll need complex numbers:
Complex[][] results = new Complex[amountPossible][];

//For all the chunks:
for(int times = 0;times < amountPossible; times++) {
   Complex[] complex = new Complex[Harvester.CHUNK_SIZE];
   for(int i = 0;i < Harvester.CHUNK_SIZE;i++) {
      //Put the time domain data into a complex number with imaginary part as 0:
      complex[i] = new Complex(audio[(times*Harvester.CHUNK_SIZE)+i], 0);
```

```
    }
    //Perform FFT analysis on the chunk:
    results[times] = FFT.fft(complex);
}

//Done!
```

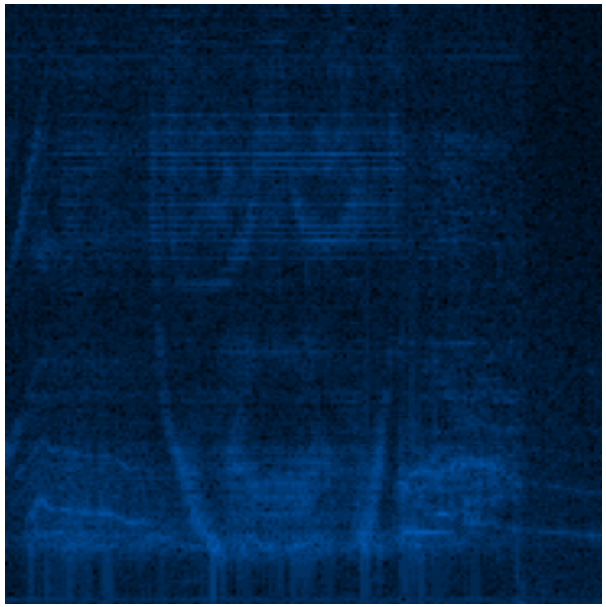
Now we have a double array containing all chunks as Complex[]. This array contains data about all frequencies. To visualize this data I decided to implement a full spectrum analyzer (just to make sure I got the math right).
To show the data I hacked this together:

```
for(int i = 0; i < results.length; i++) {
    int freq = 1;
    for(int line = 1; line < size; line++) {
        // To get the magnitude of the sound at a given frequency slice
        // get the abs() from the complex number.
        // In this case I use Math.log to get a more managable number (used for color)
        double magnitude = Math.log(results[i][freq].abs()+1);

        // The more blue in the color the more intensity for a given frequency point:
        g2d.setColor(new Color(0,(int)magnitude*10,(int)magnitude*20));
        // Fill:
        g2d.fillRect(i*blockSizeX, (size-line)*blockSizeY,blockSizeX,blockSizeY);

        // I used a improviced logarithmic scale and normal scale:
        if (logModeEnabled && (Math.log10(line) * Math.log10(line)) > 1) {
            freq += (int) (Math.log10(line) * Math.log10(line));
        } else {
            freq++;
        }
    }
}
```

Introducing, Aphex Twin

This seems a bit of OT (off-topic), but I'd like to tell you about a electronic musician called Aphex Twin (Richard David James). He makes crazy electronic music… but some songs have an interesting feature. His biggest hit for example, Windowlicker has a spectrogram image in it.
If you look at the song as spectral image it shows a nice spiral. Another song, called 'Mathematical Equation' shows the face of Twin! More information can be found here: Bastwood - Aphex Twin's face.

When running this song against my spectral analyzer I get the following result:



Not perfect, but it seems to be Twin's face!

Determining the key music points

The next step in Shazam's algorithm is to determine some key points in the song, save those points as a hash and then try to match on them against their database of over 8 million songs. This is done for speed, the lookup of a hash is O(1) speed. That explains a lot of the

awesome performance of Shazam!

Because I wanted to have everything working in one weekend (this is my maximum attention span sadly enough, then I need a new project to work on) I kept my algorithm as simple as possible. And to my surprise it worked.

For each line the in spectrum analysis I take the points with the highest magnitude from certain ranges. In my case: 40-80, 80-120, 120-180, 180-300.

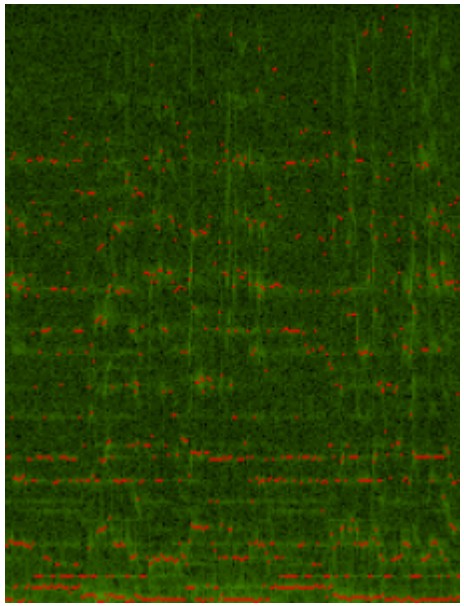
```
//For every line of data:

for (int freq = LOWER_LIMIT; freq < UPPER_LIMIT-1; freq++) {
   //Get the magnitude:
   double mag = Math.log(results[freq].abs() + 1);

   //Find out which range we are in:
   int index = getIndex(freq);

   //Save the highest magnitude and corresponding frequency:
   if (mag > highscores[index]) {
      highscores[index] = mag;
      recordPoints[index] = freq;
   }
}

//Write the points to a file:
for (int i = 0; i < AMOUNT_OF_POINTS; i++) {
   fw.append(recordPoints[i] + "\t");
}
fw.append("\n");

// ... snip ...
```

```
public static final int[] RANGE = new int[] {40,80,120,180, UPPER_LIMIT+1};

//Find out in which range
public static int getIndex(int freq) {
   int i = 0;
   while(RANGE[i] < freq) i++;
      return i;
   }
}
```

When we record a song now, we get a list of numbers such as:

```
33  56  99  121 195
30  41  84  146 199
33  51  99  133 183
33  47  94  137 193
32  41  106 161 191
33  76  95  123 185
40  68  110 134 232
30  62  88  125 194
34  57  83  121 182
34  42  89  123 182
33  56  99  121 195
30  41  84  146 199
33  51  99  133 183
33  47  94  137 193
32  41  106 161 191
33  76  95  123 185
```

If I record a song and look at it visually it looks like this:



(all the red dots are ‘important points’)

Indexing my own music

With this algorithm in place I decided to index all my 3000 songs. Instead of using the microphone you can just open mp3 files, convert them to the correct format, and read them the same way we did with the microphone, using an AudioInputStream. Converting stereo music into mono-channel audio was a bit trickier then I hoped. Examples can be found online (requires a bit too much code to paste here) have to change the sampling a bit.

Matching!

The most important part of the program is the matching process. Reading Shazams paper they use hashing to get matches and the decide which song was the best match.

Instead of using difficult point-groupings in time I decided to use a line of our data (for example “33, 47, 94, 137”) as one hash: 1370944733 (in my tests using 3 or 4 points works best, but tweaking is difficult, I need to re-index my mp3 every time!)

Example hash-code using 4 points per line:

```
//Using a little bit of error-correction, damping
private static final int FUZ_FACTOR = 2;

private long hash(String line) {
   String[] p = line.split("\t");
   long p1 = Long.parseLong(p[0]);
   long p2 = Long.parseLong(p[1]);
   long p3 = Long.parseLong(p[2]);
   long p4 = Long.parseLong(p[3]);
   return  (p4-(p4%FUZ_FACTOR)) * 100000000 + (p3-(p3%FUZ_FACTOR)) * 100000 + (p2-(p2%FUZ_FACTOR)) * 100 + (p1-(p1%FUZ_FACTOR));
}
```

Now I create two data sets:

- A list of songs, List (List index is Song-ID, String is songname) \- Database of hashes: Map<Long, List>

The long in the database of hashes represents the hash itself, and it has a bucket of DataPoints.

A DataPoint looks like:

```
private class DataPoint {

   private int time;
   private int songId;

   public DataPoint(int songId, int time) {
      this.songId = songId;
      this.time = time;
   }

   public int getTime() {
```

```
        return time;
    }
    public int getSongId() {
        return songId;
    }
}
```

Now we already have everything in place to do a lookup. First I read all the songs and generate hashes for each point of data. This is put into the hash-database.
The second step is reading the data of the song we need to match. These hashes are retrieved and we look at the matching datapoints.

There is just one problem, for each hash there are some hits, but how do we determine which song is the correct song..? Looking at the amount of matches? No, this doesn't work…
The most important thing is timing. We must overlap the timing…! But how can we do this if we don't know where we are in the song? After all, we could just as easily have recorded the final chords of the song.

By looking at the data I discovered something interesting, because we have the following data:

- A hash of the recording
- A matching hash of the possible match
- A song ID of the possible match
- The current time in our own recording
- The time of the hash in the possible match

Now we can substract the current time in our recording (for example, line 34) with the time of the hash-match (for example, line 1352). This difference is stored together with the song ID. Because this offset, this difference, tells us where we possibly could be in the song.
When we have gone through all the hashes from our recording we are left with a lot of song id's and offsets. The cool thing is, if you have a lot of hashes with matching offsets, you've found your song.

The results

For example, when listening to The Kooks - Match Box for just 20 seconds, this is the output of my program:

```
Done loading: 2921 songs

Start matching song...

Top 20 matches:

01: 08_the_kooks_-_match_box.mp3 with 16 matches.
02: 04 Racoon - Smoothly.mp3 with 8 matches.
03: 05 Röyksopp - Poor Leno.mp3 with 7 matches.
04: 07_athlete_-_yesterday_threw_everyting_a_me.mp3 with 7 matches.
05: Flogging Molly - WMH - Dont Let Me Dia Still Wonderin.mp3 with 7 matches.
06: coldplay - 04 - sparks.mp3 with 7 matches.
07: Coldplay - Help Is Round The Corner (yellow b-side).mp3 with 7 matches.
08: the arcade fire - 09 - rebellion (lies).mp3 with 7 matches.
09: 01-coldplay-_clocks.mp3 with 6 matches.
10: 02 Scared Tonight.mp3 with 6 matches.
11: 02-radiohead-pyramid_song-ksi.mp3 with 6 matches.
12: 03 Shadows Fall.mp3 with 6 matches.
13: 04 Röyksopp - In Space.mp3 with 6 matches.
14: 04 Track04.mp3 with 6 matches.
15: 05 - Dress Up In You.mp3 with 6 matches.
16: 05 Supergrass - Can't Get Up.mp3 with 6 matches.
17: 05 Track05.mp3 with 6 matches.
18: 05The Fox In The Snow.mp3 with 6 matches.
19: 05_athlete_-_wires.mp3 with 6 matches.
20: 06 Racoon - Feel Like Flying.mp3 with 6 matches.

Matching took: 259 ms

Final prediction: 08_the_kooks_-_match_box.mp3.song with 16 matches.
```

It works!!

Listening for 20 seconds it can match almost all the songs I have. And even this live recording of the Editors could be matched to the correct song after listening 40 seconds!

Again it feels like magic! :-)

Currently, the code isn't in a releasable state and it doesn't work perfectly. It has been a pure weekend-hack, more like a proof-of-concept / algorithm exploration.

Maybe, if enough people ask about it, I'll clean it up and release it somewhere.

Update:

The Shazam patent holders lawyers are sending me emails to stop me from releasing the code and removing this blogpost, read the story here.