A buffering technique for real-time display of spectrogram
Wen X.

Introduction

This document describes a programming technique for implementing spectrogram display. The spectrogram is currently a most popular way for displaying audio contents in a time-frequency plane. Usually the audio content is divided into *frames*. The centres of the frames distribute uniformly on the time axis. Frames may overlap or not. It is adequate to use 50% overlapping frames (i.e. the frame hop size being half the frame size) if a window function which tapers off at both ends is used for calculating the spectrogram. Higher overlapping rate is also used, especially in overlap-add audio manipulations. However, the high overlapping rate does not necessarily improve the spectrogram display, as the resolution is determined by the size of the window.

The displaying of spectrogram involves two stages: a calculating stage and a drawing stage. The calculating stage computes the spectrogram using FFTs. The drawing stage converts the spectrogram matrix into a bitmap of X*Y pixels. Each pixel corresponds to a (t, f) pair. The colour of this pixel is mapped from the point (t, f) in the spectrogram matrix through a palette. However, the spectrogram matrix is sampled in time at frame centres and in frequency at bin centres. It rarely happens that t falls exactly at a frame centre or f at a bin centre. There are multiple ways to deal with this. In our implementation a continuous "spectrogram" is interpolated from the spectrogram matrix first in time using a combined linear-parabolic interpolation, then in frequency using linear interpolation. In this case to compute the colour for a pixel 4 to 6 points from the spectrogram matrix is required, involving two frames. However, if the number of frames is more than the number of pixels ("dense spectrogram"), the interpolation in time becomes less meaningful. In this case only the frame whose centre is closer to *t* is used.

The problem addressed in this document arises from the experience of waiting for minutes for the spectrogram of a long audio excerpt to be displayed, such as in an audio editor. What happens here is that the program usually calculates the spectrogram of the whole audio content before displaying it. The FFT computation is generally much faster than real time, but to compute 100,000 frames at a time is another matter. The user, however, is likely to become upset for the waiting.

The key question here is, do we need for calculate 100,000 frames of FFT just to show the spectrogram? Of course we do if the displaying device holds 50,000 pixels along the time axis, but that is the high-end equipment. The typical case is that the spectrogram is drawn onto a canvas of, say 600~1280 columns / rows. Accordingly no more than 1200~2560 frames need to be calculated to show the spectrogram. Given the size of the canvas and the duration (i.e. a starting point and an end point) of the audio content being drawn, the frames that are needed are determined. This is the starting point of the buffering technique presented in the document.

One last comment before moving on to the technical part: it is possible not using any buffering at all if one can afford calculating all the necessary frames (as we've said, up to 2560 frames) each time the spectrogram is drawn. One needs to store the spectrogram only when the drawing stage is separated from the calculating stage, so that when the drawing method is invoked the calculations are not repeated.

The buffering arrangement

Let the length of the audio content be L (from 0 to L-1), the window width be W, L>>W, and the hop size be H<W. We align the start of the first frame to 0, therefore it is centred at W/2. The total number of complete frames is floor((L-W)/H+1). The total number of frames including the incomplete ones is ceil(L/H). floor() and ceil() are downward and upward rounding methods. Let the total number of frames be FR. The fr^{th} frame starts from fr*H, and terminates at fr*H+Wid-1.

The main idea of the buffering is to store the already calculated spectra and reuse them when they are needed in the future. In the old way the whole spectrogram is computed and stored in a matrix. However, since now we compute only the necessary frames, there is no immediate need for allocating buffers for the complete spectrogram of FR frames, since only the spectra of the calculated frames are stored.

Let the physical memory be referred to by Buffer[][], where Buffer[x][:] stores the spectrum of a frame, which is an array of size W. We introduce two extra buffers: a buffer x[] of size FR where x[fr] records the position of frame fr in the physical Buffer, and a buffer v[] of size FR where v[fr] records whether the spectrum of the frame fr is computed and ready for use. x and v may be treated as separated physical memories or combined together with v occupying only one bit for each frame. The final result of this arrangement is that if x[fr] is a valid frame index then the frame fr has a place in

Buffer; if at the same time v[fr] is valid, then Buffer[x[fr]][:] is the ready-for-use spectrum of the frame fr.

There is no constraint on how the physical Buffer shall be acquired. One may acquire one frame each time a new frame is to be added, or acquire the space for many frames for future allocation. Each frame occupies only one frame buffer in Buffer. The size of Buffer never grows to more than FR frames.

The above arrangement is summarized as follows:

int FR; //number of frames in audio
int Count; //number of frames allocated in Buffer
int x[FR]; //pointer to frames in Buffer
bool v[FR]; //validness tag of frames in Buffer
float** Buffer. //pointers to the physical memory(s)

Basic operations

To initialze

The buffer is initialized after FR is determined by the following sequence:

Allocate x and v;
x[fr]←-1, v[fr]←false, fr=0, 1, …, FR-1;
Count←0.

To allocate physical memory for a frame fr

If x[fr]<0, then x[fr]←Count, v[fr]←false, Count++, (acquire space Buffer if necessary).

To load the spectrum of frame fr

If x[fr]<0, then x[fr]←Count, Count++, (acquire space for Buffer if necessary);
Load Buffer[x[fr]][:];
v[fr]←true.

To use the spectrum of frame fr

If x[fr]<0, x[fr]←Count, Count++, (acquire space for Buffer if necessary), v[fr]←false;
If v[fr]=false, compute the spectrum of frame fr and save it to Buffer[x[fr]][:], v[fr]←true;
Use Buffer[x[fr]][:].

To clear

x[fr]←-1, v[fr]←false, fr=0, 1, …, FR-1;
Count←0.

To clean up

Free all physical memories allocated for Buffer;
Free x and v.

Use for spectrogram display

Let the displaying duration of the audio starts from T1 and ends at T2-1, and the time axis on the canvas be X pixels long, ranging from 0 to X-1. Then a pixel x corresponds to the time

$$t=T1+(x+0.5)*(T2-T1)/X.$$

The two frames involved for calculating the pixels at x are located by

$$fr1=floor(t-W/2)/H,\ fr2=ceil(t-W/2)/H.$$

In the case fr1=fr2, only one frame, and no interpolation in time, is needed. In the case of dense spectrogram, we choose fr1 or fr2 by comparing t-(W/2+fr1*H) and (W/2+fr2*H)-t and choose the smaller one.

The buffers are initialized on drawing the first spectrogram, and freed and re-initialized each time the audio content is completely changed or the spectrogram calculation settings, such as window shape/width/hop size are changed. To following sequence are used for drawing the spectrogram.

For x=0, 1, …, X-1, do 1~6:
1. calculate fr1 and fr2 if applicable;
2. if x[fr1]<0, x[fr1]←Count, Coun++ (acquire space for Buffer if necessary), v[fr1]←false;
3. If v[fr1]=false, compute the spectrum of frame fr1 and save it to Buffer[x[fr1]][:], v[fr1]←true;
4. interpolate Buffer[x[fr1]][:] in frequency to fit the pixels;
5. do 2~4 for frame fr2 if applicable;
6. calculate all pixels at x on the time axis.

Additional buffering may be applied to store the interpolation result in 4 for reuse at different x's. By drawing the spectrogram in this way, we make sure that

1. only necessary frames of the spectrogram are computed;
2. once computed, no frame of the spectrogram is re-computed as long as it remains valid;
3. no more than the necessary buffers to store the already-calculated results are allocated (although limited amount of physical space may be acquired in advance).

Local change of audio content

The local change of audio content is common in audio editors. By "local" we mean that the change does not cover the whole duration of the audio content, and that the untouched parts remain where they are with no time shifting. Local changes of audio content require that re-calculation of part of the spectrogram. This is done by invalidating certain frames. Let the change of content happen in the duration from T1 to T2. The first and last frames affected are determined by

$$fr1=ceil((T1-W+1)/H),\ fr2=floor(T2/H).$$

To invalidate the affected frames, we do the following:

For fr=fr1, …, fr2, do 1:
1. if x[fr]≥0 and v[fr]=true, then v[fr]←false.

The physical space of invalidated frames is kept for these frames even if their spectra are no longer used. However, the possibility of these frames being used in the future is higher than previously unused frames, since where the content has been altered is very likely to be the part of interest.

Time shifting of whole hops

Time shifting mostly happens in two cases: realignment and cutting. Generally speaking there are no good ways to reuse already-calculated spectrum after the waveform audio has been time-shifted. When realignment takes place the spectra calculated from the whole realigned part is invalidated. When there is a cutting operation either this side or that side (or both) of the cut is realigned and the spectra involved are invalidated. However, if the time shift is a multiple of the frame hop, most frames of the realigned part are reusable. It is especially useful if the application forces the involved time shifting to be multiples of the frame hop.

In the case of a time shift of M hops, let the unaffected frames be between fr1 and fr2 in the original content. As a result of the time shift, they now become the fr1+M and fr2+M. This is easily implemented for M<0 by the following.

For fr=fr1, …, fr2, do 1:
1. if x[fr]≥0 and v[fr]=true, then x[fr]↔xr[fr+M], v[fr+M]←true.

For M>0 this becomes

For fr=fr2, …, fr1, do 1:
1. if x[fr]≥0 and v[fr]=true, then x[fr]↔xr[fr+M], v[fr+M]←true.

All frames affected by the time shifting other than fr1+M … fr2+M shall be invalidated.