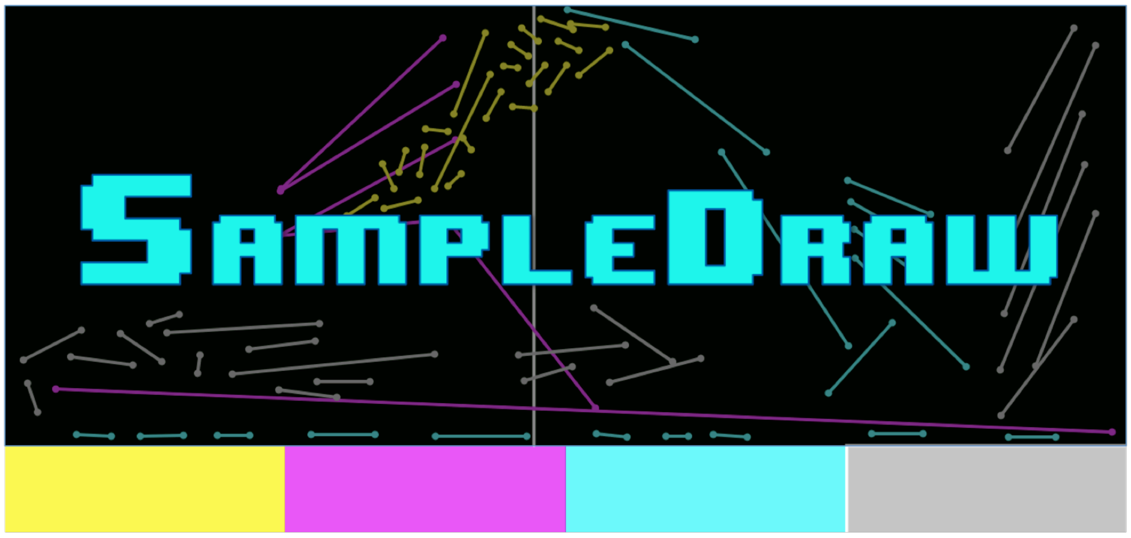
Music and Audio Programming
ECS7012
Final project

Teodoro Dannemann



In this document, we present the design, development and evaluation process of SampleDraw, a tool that allows the visual manipulation of samples in the time-pitch domain. Firstly, users upload their custom samples to Bela. Then, through Bela GUI, users can draw lines in a time-pitch space, where each line corresponds to a specific sample played back. Samples are manipulated in the time domain for changing their duration and pitch according to the drawn lines. We used a simple method based on circular buffers that allows both time-scale modification and also pitch-scale modification of the samples. The tool allows the simultaneous playing of several samples at the same time, and it allows the user to upload up to four different custom samples.

1. Introduction

Music sampling has shown to be a useful technique in the composition process, that can go from audio collages to grain synthesis [1], [2]. Normally, these tools use a visual representation of sounds – such as spectrograms, for instance - and they allow users to directly manipulate specific sound features to trim, stretch, compress, merge, pitch and combine several sounds to create a music composition.
Bela's low latency feature plus its GUI library, make it ideal for an implementation of a graphical user interface that allows sample collages. We propose here a first simple step, where users can draw coloured straight lines in a blank canvas, each line colour corresponding to a different audio sample uploaded by the user. The canvas here represents a time-pitch space, so each line is decomposed in its *x* component length that will constitute the duration of the corresponding sample, while its *y* component will map to the instantaneous pitch of the sample. Thus, samples are stretched/compressed (i.e. time-scale modification) but also raised/lowered in pitch (i.e. pitch scaling) in order to match user temporal drawn patterns. Interestingly, we can obtain both time-scale and pitch-scale modifications by using the same method, which basically consist in a resampling plus a time-modification method.
In the following, we will start describing a series of state-of-the-art algorithms related to time-scale and pitch-scale modifications. We will then present our method, which integrates both of them in a straightforward way, by using a circular buffer. We finally show an evaluation of the performance of our algorithm, by measuring roughly the estimated processing time for different setups.

2. Related Work

2.1 Pitch-scale modification methods

It is well known that when we change the sample rate of an audio signal we alter the original pitch of it. However, not only the pitch, but also the length of the audio sample changes. For instance, if we have a one-second length sound that is sampled at a rate of 44.1 kHz, if we resample it at the half sample rate (22.05 kHz), and play it back, then the resulting sound frequencies will halve and the resulting duration will be the double. Thus, if we want to pitch shift a sample and maintain its original duration, we need to firstly resample it and then apply a time-scaling procedure to get the sample back to its original duration. This is depicted by figure 1 (taken from [3]), where an original sample rate f_s is resampled at a frequency of f_{rs}. This produces a time-scale modification by a ratio of f_s/f_{rs}. Then, for going back to the original sample duration, a Time-scale modification algorithm must scale the resulting sound by the inverse factor, f_{rs}/f_s.

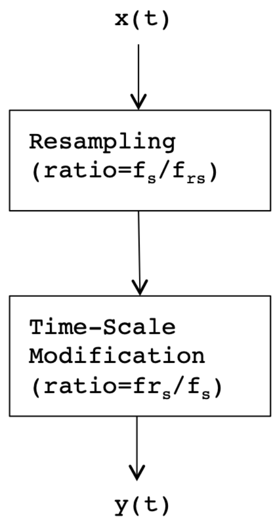

x(t)
Resampling (ratio=f_s/f_{rs})
Time-Scale Modification (ratio=fr_s/f_s)
y(t)

Figure 1: Usual two-step approach to pitch-shifting a signal. First we resample, and then we make a time-scale modification to take to the original duration of the sample

We have to emphasise that these methods are constrained to the time-domain. Although there are also interesting pitch shifting methods in the frequency domain (e.g. the *phase vocoder* [4]), we will here focus on time domain algorithms, as they are less computationally expensive and thus more suitable for real-time contexts [5]. The drawback, however, is that they are less suitable for complex signals with a lot of non-harmonic components [3].

2.2 Time-scale modification methods

The traditional approach for time-scale modification (TSM) is the decomposition of the sound wave into several audio "chunks" of very short duration (around 50-100 ms). These frames are not disjoint, but they rather overlap, so we can rearrange them (joining or separating them) in order to obtain the desired TSM. These leads to a family of algorithms, being the most simple case of them the OLA (Overlapping Add) method and all its variants (PSOLA, WSOLA, see [6] with a complete review).

However, here we will follow a different approach inspired in the method proposed by Haghparast et al [3], which, in turn, is inspired in [7]. Interestingly, in this approach, both resampling and TSM are put together in the same algorithm. The method uses a circular (or ring) buffer where the read pointer is allowed to move in a different speed than the write pointer. This leads to the obvious problem that both pointers could cross each other, which would cause clipping. In order to avoid this, an overlap zone around the write pointer is considered. If the read pointer approaches this zone, then it will be moved back a certain amount, say *L*, of samples. This is depicted in figure 2. Let's suppose that the write pointer (blue arrow) moves one frame per time step (blue dashed line), and that the read pointer (red arrow) moves faster than this (red dashed line), say Δr frames per time step (this means we are increasing the pitch of the original sound). The overlap area is represented by the blue arc attached to the write pointer. For

T=t and T=t+1 the read pointer simply jumps forward and so does the write pointer. However, for T=t+2 the read pointer is already in the overlapping zone, so it must jump back *L* frames.
The latter means that several samples will be read more than once, so it will compensate and fill the gaps of samples that the reader misses as it is taking jumps in between them.

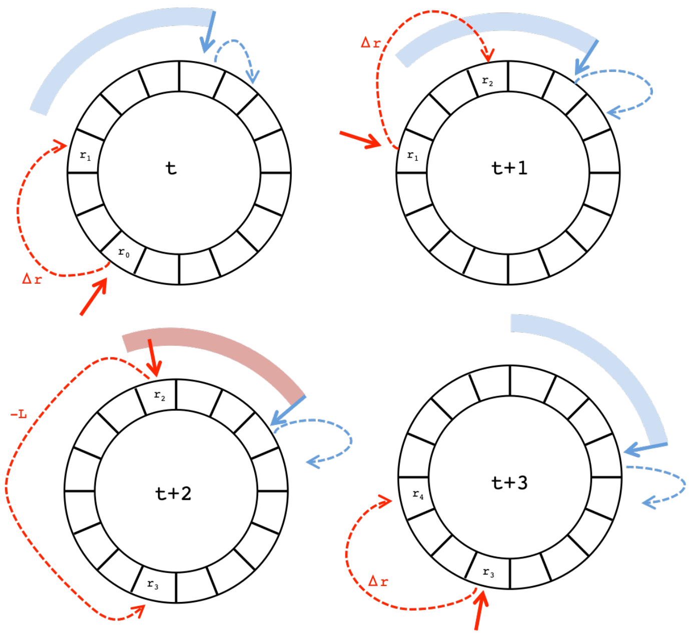


Figure 2: depiction of how the ring buffer method works. The read pointer makes jumps of length Δr. If it lies on the overlap area, then it must jump back *L* sites. Image is an adaptation from figure 3 of [3]

Notice that the jump length, Δr, corresponds with the pitch-scaling factor. For example, if Δr=1, then both read and write pointer will move at the same speed and the sound remains unaltered. If Δr=2, instead, then the read pointer will miss the half of the samples, so the resulting sound will double the pitch. However, the duration of the resulting sound will be the same (although the read pointer reads half of the samples when passing, it makes twice the passes, as it jumps back).

3. Design

For our work, we propose two main modifications to the ring buffer method:

1. Following what has been proposed in [8], we use two read pointers that are distanced 180 degrees in the circular buffer, i.e. their distance is always constant and equal to the half of the length of the buffer. We also add a crossfade that will softly transition across the two read pointers, so when one of them is in the overlap zone, the crossfade will “mute“ that read pointer

and activate the second read pointer, and vice versa. In this way, we can dispense with the necessity of the reader to make jumps backwards.

2. Notice that this method only changes the pitch of the sound, but not its duration. In order to do that, we add a resampling procedure before our circular buffer. Of course, as the resampling will shift the pitch, through our circular buffer we will counterbalance this effect.

A diagram of the system is shown in figure 3 (left side). Consequently, as shown in figure 3 (right side), if we have a sound sample of duration and frequency (D_1,f_1) (let's assume it is a single frequency sound as a sine wave oscillator), and we want to transform it into a new duration and frequency values (D_2,f_2), then, our method will proceed as the following. Firstly, by resampling by a factor D_1/D_2, we will obtain an output of duration D_2, as desired, but its frequency will equal to f_1D_1/D_2. Therefore, we use the circular buffer to map the frequency to the desired one. This means that the circular buffer switches the frequency by a factor of f_2D_2/f_1D_1. Finally, the output of the circular buffer will have the desired duration and frequency.

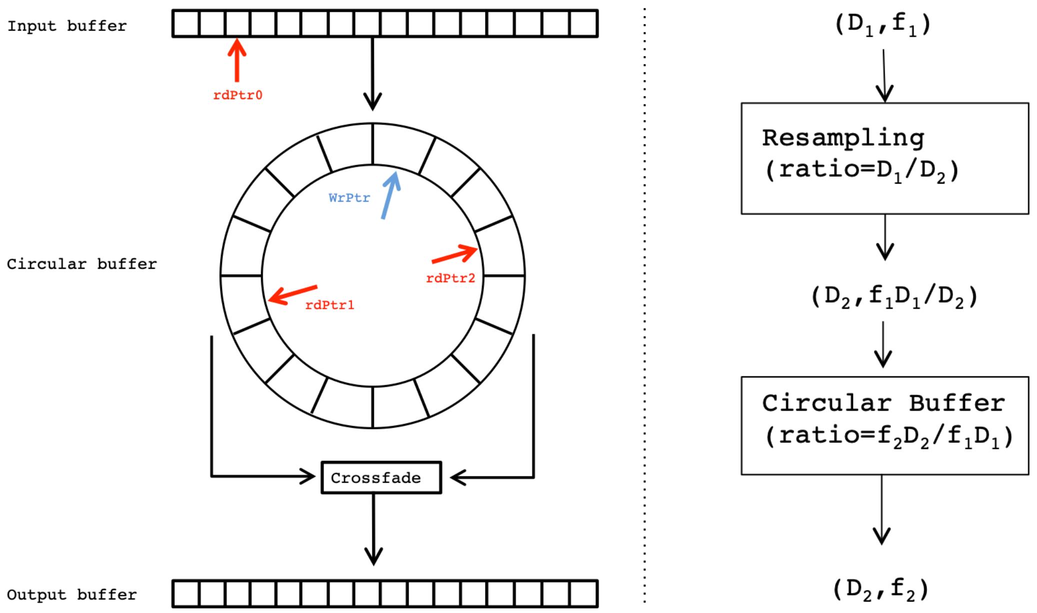


Figure 3: Left: diagram of the circular buffer method and the corresponding read and write pointers. Right: Diagram of the mappings from the original values (D_1,f_1), into the new desired values (D_2,f_2)

4. Implementation

4.1 GUI interaction and communication with Bela

User's main interaction occurs in the GUI[1]. After they have loaded four custom sounds to the Bela project, they are prompted to the GUI, where they can start "drawing sounds". Figure 4 shows a capture of the GUI display, with an example interaction. Users can add lines of four different colours, which are picked by clicking each rectangle in the lower bar. A white bar (vertical line in the centre) continuously moves from left to right, acting as a play head.

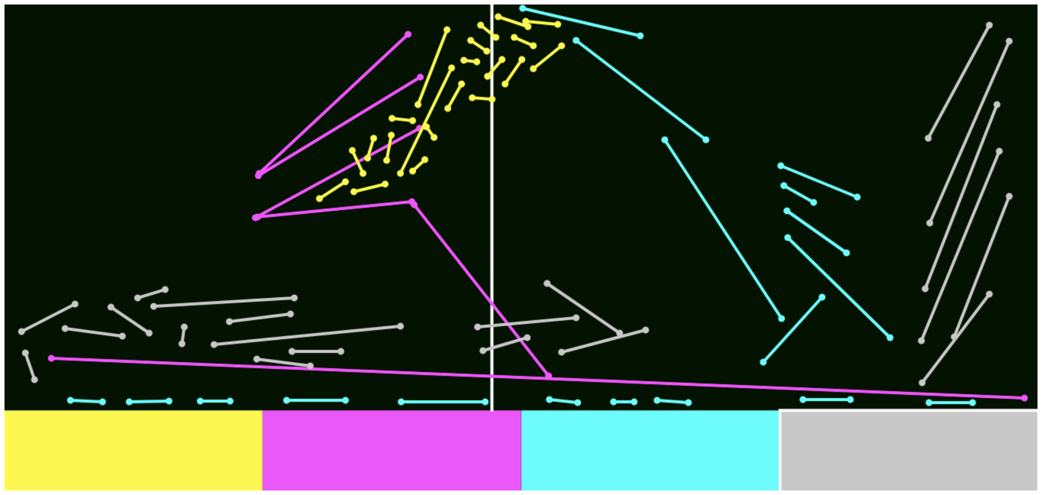


Figure 4: Screen capture of the GUI. Each colour corresponds to a sound sample.

Each time users add a new line segment — we will call it a stroke from now on - the features corresponding to this stroke are sent to Bela where they are added to a matrix. The sent data corresponds to an array containing a unique id assigned to that stroke, the (x,y) coordinates of the two ends of the stroke (in relative coordinates), and the stroke type (colour) number. At the same time, the GUI continuously receives from Bela the value of current time, and it updates the play head (white vertical bar) to the corresponding position. As soon as the play head touches a stroke, the sound sample corresponding to that stroke will start playing, and it will do so as long as the play head still touches that stroke. We will explain later how do we achieve this with

[1] GUI implementation was a modification of an already created project. However, the data structures that are sent to Bela were a new development for this project. In other words, we recycled only the visual display of the canvas, but the objects that are added by the user and how are they manipulated and sent to Bela is a whole new work. You can see the original project here: https://teo523.github.io/sonicDraw/public/

Bela. When the play head reaches the right end of the canvas, it starts from the left again, so the sound is continuously looped.

4.2 Storing, activating and deactivating strokes

In Bela, we created an object called *Strokes* for storing all the received strokes. *Strokes* is a vector of vectors (i.e. a vector that contains a vector in each position).
Bela continuously reads the last buffer (stroke) sent from the GUI. Each new stroke is received as an array, which is added as a new vector to *Strokes.* A unique id for each stroke avoids that they are added more than once to *Strokes.* We add an additional boolean variable to each stroke that will indicate whether that stroke is active or inactive (i.e. if the play head is passing through that specific stroke).
On the other hand, we created a global time counter that goes from zero to the total duration of the loop (set to 10 seconds by default, but it can be easily modified in the code). The time counter is updated for each execution of render. For each value of the time counter, we firstly check for strokes that must be activated. The condition for doing so is that the time counter is greater than the minimum x coordinate of the stroke, and smaller than the maximum x coordinate of the stroke (i.e. that the play head is touching that stroke). Then, we deactivate strokes that were active in the previous time step and that now don't fulfil the above requirement to be active. Finally, we write to the output the values of the active strokes. How to calculate these values will be explained in the following section.

4.3 Mapping strokes to sounds

We created an array of N circular buffers of equal size (N=10 by default). Each time a stroke is activated, we look for an available circular buffer, and we assign it to the stroke. This assignment is made by an auxiliary array that stores which stroke is pointing that specific circular buffer. Notice that the method allows a maximum of N sounds being played simultaneously.
Once the strokes are active, we have to start playing them. However, they won't be directly played but, rather, they have to be transformed both in duration and pitch. For achieving this, each active stroke will follow the mapping given in method shown in figure 3. This means that each active stroke will have attached to it not only a circular buffer, but also a read pointer for the input sample, and two read pointers plus one write pointer for the circular buffer.
We will explain how the mapping of duration and pitch works by using an example.

Let's suppose we draw a stroke in the GUI that goes from the point with coordinates (x_0,y_0) to the point with coordinates (x_1,y_1), where coordinated are given in relative terms of the absolute height and width of the canvas, so $0 \leq x \leq 1$, and $0 \leq y \leq 1$. Then, the duration of that stroke – the time that the play head will spend touching it – will be given by $L(x_1-x_0)$, where L is the total duration of the loop (10 seconds by default). The pitch case is somewhat different, as pitch is an instantaneous feature (in contrast with duration), so it will vary along the same stroke. Given an arbitrary value of y (with $0 \leq y \leq 1$), we propose the following mapping:

$p = p_0\left(Q + \frac{1-Q}{0.5}y\right)$ (i)

where *p* is the resulting pitch, p_0 is the original pitch of the sound sample, and *Q* is a factor. Note that when $y=0.5$, then $p=p_0$. This means that when we are in the middle point of the canvas, then the sound will remain unaltered. Also, when $y=0$, then $p=Q*p_0$, so *Q* is the multiplying factor of the pitch when we are in the bottom of the canvas (thus we choose $0 \le Q \le 1$ for a positive linear relationship from height and pitch). Finally, the maximum value that can take the pitch is $p=2p_0(1-Q)$, so for small *Q,* the pitch almost doubles the original when we are in the top of the canvas.
Now, the question is how do we translate these values into actual implementations in our circular buffer (where the pitch transformation takes place). These is easily answered if we notice that, for example a doubling in pitch means that the read pointers of the circular buffer must go twice fast as the normal, as explained in section 2. Then, for instance, if we obtained from equation (i) that $p=2p_0$, then the read pointers of the circular buffer will advance two sites at a time. If we obtained, say, $p=1.5p_0$, then we advance the read pointers by 1.5 and look for the closest site (by applying the floor function).

5. Evaluation

As explained in the introduction, time-domain pitch shifting algorithms have worse quality but better performance than frequency-domain algorithms. Therefore, we propose to assess how efficient was our algorithm, and how suitable is for real-time processing. For this, we calculated the mean processing time, measured as the percentage of time that Bela takes in computing the output, when compared with the total available time. We recall that, for each sample, the maximum available time for processing in seconds, should be:

$$T_{max} = \frac{1}{sampleRate}$$

We used an empirical method, where we connected two Bela devices, the first one to apply the algorithm, and the second one to use as a scope (given the current global contingencies, we didn't account with a proper scope). The code for the scope Bela is also available in the project folder.
Mainly, the idea is that, when the render function starts executing in the first device, then it activates a GPIO output pin[2], which will be active as long as the code is executed, and we deactivate it again when the code finishes. In this way, the pin will remain UP (value = 1) approximately the duration of the execution of the code corresponding to one sample. In figure 5 we show an example reading of the scope (Bela). According to this, we are interested in calculating the ratio $r=d/T_{max}$, where *d* corresponds to the time the signal value is high.

[2] in contrast to using Bela digital I/Os, which are sampled at audio rate, we can control sychronously the GPIOs of Bel ato get an accurate and minimally invasive measures of code segment processing times, as pointed in http://docs.bela.io/synchronous-gpio_2render_8cpp-example.html#a3

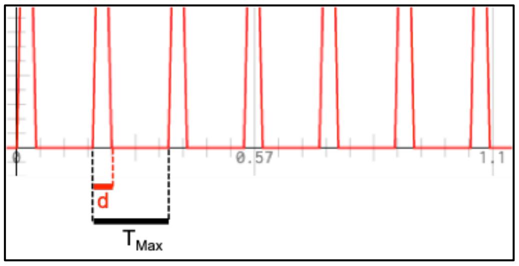


Figure 5: The processing time for each sample is given by *d*. We are interested in the ratio $r=d/T_{max}$, the percentage of time that is used in processing the code.

We evaluated *r* for different number of drawn strokes. We expected to have an increasing relationship, as more strokes means more processing, thus a greater value for *r*. We performed 100.000 simulations for each value, obtaining the average values shown in figure 6.

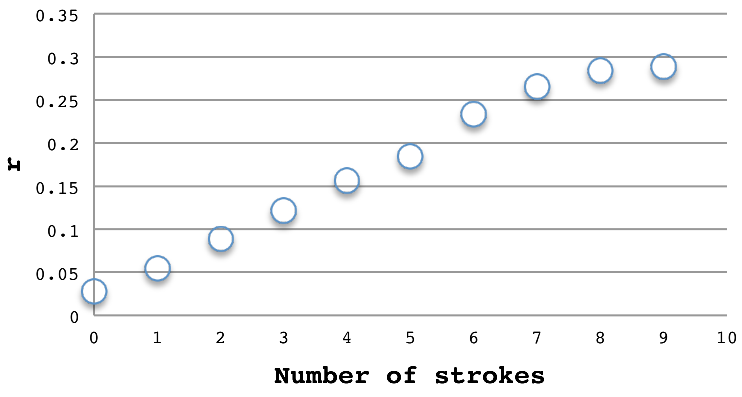
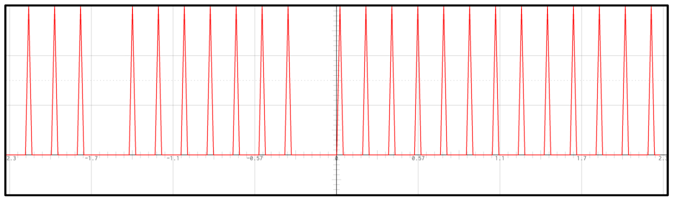


Figure 6: time processing ratio for different number of strokes drawn. As expected, more strokes mean more processing time.

We can see that, indeed, *r* increases with the number of strokes, and the increase seems to be linear (except for higher values, where the curve seems to flat). For *n=9* strokes, we get an *r* of almost 30%. Although high, we still have a 70% of free time for calculations, indicating that the algorithm could have the capacity for more strokes (we used a maximum of 10 in this case). We also show in figure 7 a screen capture of the oscilloscope for the cases *n=0* strokes (above), and *n=9* strokes (below), emphasising a marked difference in processing time.

6. Conclusions

In this report we have shown the development and assessment process of a tool that allows the creation of sound collages by using pitch and time shifting of sound samples. We used a well-known algorithm that combines resampling and time-scaling to achieve the desired mapping to arbitrary values of duration and pitch. The algorithm showed to be highly time efficient, allowing the inclusion and processing of several



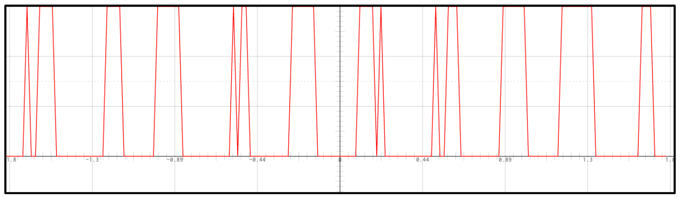


Figure 7: scope readings for *n=0* strokes (above), and *n=9* strokes (below).

samples at the same time. In our case, we tried up to 10 sounds playing at the same time (i.e. 10 available circular buffers), getting a maximum usage of 30% of the total available processing time, suggesting the possibility of extending this method to both more circular buffers but also a higher number of sounds uploaded by the user.
A drawback, however, is that the quality of the mapped sounds is not the best, especially when compared with frequency-domain based algorithms. This trade off between time efficiency and sound quality is fundamental, and the decision of which algorithm to use should be based on the specific usage but also the kinds of sounds that will be used, as our tool shows good performance quality-wise for simple sounds.

References

[1] N. Bogaards, A. Roebel, and X. Rodet, "Sound Analysis and Processing with AudioSculpt 2 To cite this version : Sound Analysis and Processing with AudioSculpt 2," *Int. Comput. Music Conf.*, no. Icmc, p. 1, 2004.

[2] L. Engeln and R. Groh, "Visualaudio-Design – Towards a Graphical Sounddesign," pp. 1–8, 2019.

[3] A. Haghparast, H. Penttinen, and V. Välimäki, "Real-time pitch-shifting of musical signals by a time-varying factor using Normalized Filtered Correlation Time-Scale Modification (NFC-TSM)," *Proc. 10th Int. Conf. Digit. Audio Eff. DAFx 2007*, pp. 7–13, 2007.

[4] J. L. Flanagan and R. M. Golden, "Phase vocoder," *Bell Syst. Tech. J.*, vol. 45, no. 9, pp. 1493–1509, 1966.

[5] U. Zölzer, *DAFX: digital audio effects*. John Wiley & Sons, 2011.

[6] J. Driedger and M. Müller, "A review of time-scale modification of music signals," *Appl. Sci.*, vol. 6, no. 2, p. 57, 2016.

[7] F. F. Lee, "Time compression and expansion of speech by the sampling method," in *Audio Engineering Society Convention 42*, 1972.

[8] YetAnotherElectronicsChannel, "Pitch Shifting - Audio DSP On STM32 (24 Bit / 96 kHz)." [Online]. Available: https://www.youtube.com/watch?v=p_a8mDcAvOg&t=65s.