Music and Audio Programming – Final Project: Pitch-Aware Granular Synthesis

Max Graf – 190120296

Abstract

We present a real time audio synthesis algorithm based on granular synthesis. Instead of creating grains from the raw input signal, we dynamically mask the audio data in in the frequency domain in order to create sub-signals, from which grains are created. This process enables any audio signal with sufficient content to be used as a basis for synthesis and results in a large variety of possible timbral characteristics. We integrate the algorithm into a digital musical instrument on the Bela platform to demonstrate its capabilities. We evaluate the system on three different types of source data and discuss the results.

Introduction

Granular synthesis describes the process of splitting an audio signal into small units of lengths ranging from 1ms to 100ms - *grains*. The technique can be employed to achieve results that are hard or impossible to do with traditional modes of synthesis (such as conventional sampling). The subdivision of the signal can be exploited to decouple two typically interconnected properties of sound: Pitch and tempo. This can be used to manipulate the tempo of a given piece of audio without altering its pitch, and vice versa. In terms of synthesis, a selection of grains can be used to create complex sonic patterns by varying their length, positions in the source material, pitch and other qualities.

The concept of subdividing an audio signal into small units in order to compress or expand its frequency information was first formulated by Gabor [1]. Xenakis subsequently formulated a theory for the use of grains of sound in composition [2], transferring the original idea from telecommunications theory to musical applications. The concept was further developed by Roads, who produced the first digital implementation of granular synthesis [3]. The first real time system based on the technique was presented by Truax [4], opening up the possibilities of musical performance to the paradigm. Following that, several extensions and adaptations of the original technique were developed, such as the use of cellular automata as a method for grain selection [5], a technique for synthesising environmental, natural sounds [6] and a system designed specifically to create sonic textures [7].

These systems share the common property of generating pitch by means of periodic repetition of grains. While this method is adequate when dealing with grains of small lengths (1ms-50ms), it may produce potentially undesirable frequency components, depending on the spectral composition of the source material, especially when dealing with longer grains. In order tackle this problem, we propose to dynamically mask the frequency representation of the source material to create a “clean” starting point for granularisation, thus allowing for accurate modelling of pitch.

Design

System overview

The proposed system is an interactive, polyphonic instrument that converts MIDI input (notes) to pitched sounds. To do that, a short window of 1.25s is extracted from the source audio data. This window is dynamic and its position can be changed in real time. Its content, however, consists not of the raw waveform data, but rather the frequency domain representation of the waveform, which is calculated using the short-term Fourier transform [8].

Upon triggering a note event, the desired fundamental frequency is calculated from the MIDI data, and used to mask the frequency components of the window. The masking process is based on the frequency bin the current fundamental frequency is most closely related to. Due to the finite frequency resolution of the discrete Fourier transform this produces some inaccuracies. However, due to the logarithmic nature of human pitch perception [11], the perceived gravity of this error decreases as pitches of notes increase.

Design process

The first step in designing the system was to research different types of granular synthesis with a special regard for their approach to pitch. As discussed above, most techniques rely on periodic repetition of grains to produce pitch, which can lead to unwanted frequency components in the output if the source material is polyphonic in content. For example, if the source audio data contains a perfect fourth, single notes played will produce perfect fourths for whatever base frequency the incoming notes are converted to. This may be negligible for grains with very short durations, but it becomes audible once the grain length passes the amount of time it takes the two combined frequency components to finish one period. For example, a recording of a piano playing a perfect fourth consisting of C4 at 261.63Hz and F4 at 349.23Hz, will produce sounds with more than one perceived pitch if subjected to granular synthesis, given that the grain length is greater than 3.8ms, assuming a sampling rate of 44100Hz. This can grain length is calculated by first obtaining the duration for one period of the lower of the two frequencies, given the sampling rate

$$T_s = \frac{F_s}{F_n} = \frac{44100}{261.63} = 168.56 \qquad \textit{Eq. 1}$$

Where F_s is the sampling rate of the system, F_n is the fundamental frequency of the lower note and T_s is the time of one period in samples. Using T_s, the maximum grain duration in seconds is then calculated by

$$D_g = \frac{T_s}{F_s} = \frac{168.56}{44100} = 0.00382 \qquad \textit{Eq. 2}$$

This duration of 3.8ms (in the case of C4 and F4) severely limits creative possibilities in terms of synthesising controlled polyphonic sounds from polyphonic source material, as the grain length can never be increased beyond this point without perceiving more than one fundamental frequency component when playing only a single note. Naturally, the value of D_g decreases even further, as the fundamental frequencies of the source material increase. In practice, that means that controlled, pitch-accurate, polyphonic granular synthesis is only feasible if the source material is monophonic. The presented system aims to fill this particular gap. More specifically, it is designed to guarantee monophonic sounds for any single given note, independent of the source material. Thus it is aimed primarily at the musical aspect of granular synthesis, with its primary focus being the use as a musical instrument.

The original idea for the project was to perform an analysis of fundamental frequencies of the source material in order to create an index identifying grains by their pitch. This was altered to the current solution however, as it did not solve the problem of multiple perceived pitches in polyphonic sources.

Instead, the approach was taken to mask the input signal dynamically on demand (i.e. for every *note-on* event), in order to guarantee a sufficiently accurate approximation of a single correct pitch. This was achieved by the aforementioned windowing of the source, and the subsequent creation of the frequency domain representation using the short-term Fourier transform. As is common in digital instruments, and synthesisers in particular, the concept of *voices* was used to abstract the process of synthesis for different notes. Here the voices are encapsulated audio generation units that provide their own buffers and methods to convert the base frequency domain representation, the *grain source buffer*, into masked, time-domain signals ready for playback. As such, every voice creates and contains its own set of grains. While the control of hyperparameters of grains, such as the base value for grain lengths and the number of grains currently present in a voice, are globally controlled, their integration into voices allows for independent control of more subtle parameters, such as the starting positions in the source buffer and small changes in their lengths. This allows for the system to be used both as a "clean" instrument with predicable outcomes, as well as a more experimental device that can be configured to create more chaotic results.

The grains themselves are subject to a separate windowing process to further shape their sonic qualities. Four different window functions were chosen to provide accurate control over their volume envelope. Diagrams of the shape of these window functions are shown in table 1. The x-axes represent the current durations of the grains to which the window is applied. The y-axes indicate the values of the volume envelope applied to the grains, in the range [0...1].

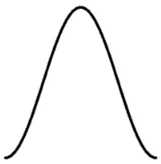
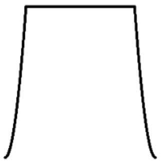
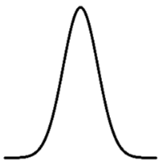
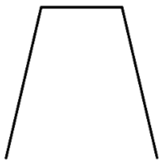
			
Hann	Tukey	Gaussian	Trapezoidal

Table 1: Visualisation of the four provided window functions

To give further control over the shape the sound, two digital filters were added to the end of the signal chain: A low-pass filter to globally reduce high frequency content and a high-pass filter to attenuate potentially unwanted low-end frequencies.

Finally, to streamline control over the system, a graphical user interface (GUI) was implemented in the *p5 js* framework[1], using Bela's supplied communication system. All controllable parameters are exposed in the GUI, enabling real time interaction with the system during performance.

[1] https://p5js.org/, accessed on 11/05/2020

Details of implementation

This section describes the functionality of the system's components in detail. The baseline code for the implementation of this project was adapted from the *FFT-phase-vocoder* example[2], provided publicly within the Bela framework. Specifically, the loading of audio data into Bela's working memory in the `main.cpp` file was adapted from the example project to support pre-loading of more than one audio file. Furthermore, the initialisation and memory allocation routine for the short-term Fourier transform fields was adapted from the example project's `render.cpp` file. The `SampleData` structure was adopted to simplify the handling of waveform sample data. A high-level schematic overview of the system's signal chain is given in figure 1. It should be noted that the steps occuring between the triggering of notes and the final filtering are carried out separately for *each active voice*.

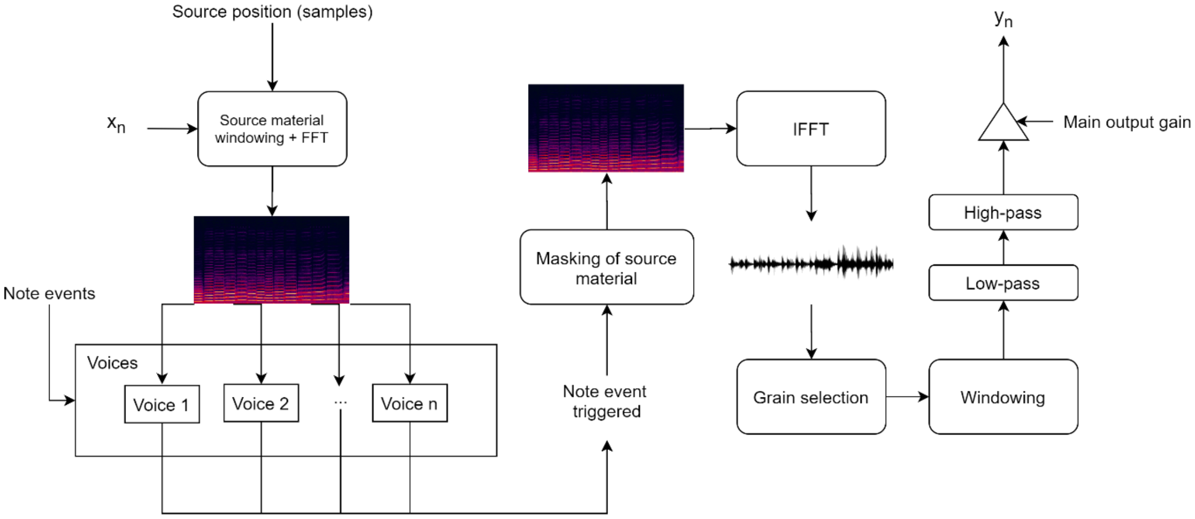


Figure 1: Schematic overview of the system's signal chain

Render

The `render.cpp` file describes the central component of the system. It connects the individual subcomponents and creates a continuous output audio stream by combining individual playing voices and applying the two filters to the output signal. Its fields include MIDI data handlers to trigger and release voices, pointers to the source audio files' waveform data, data structures to keep track of active voices, as well as routines for interacting with the user interface. During the `setup()` phase of the program three previously loaded (see `main.cpp`) raw waveform **.wav* files are exposed to the system via its working memory, all necessary memory for audio buffers, Fourier transform utilities, and further fields is allocated, and the connection to the GUI is established. The NE10[3] library was used to perform the Fourier transform calculations.

On each render pass, parameter information is retrieved from the GUI and applied to the system. Then the data structure containing voice information is queried for active voices, and those active voices are queried for their next audio sample. Listing 1 shows one of the most important functions of the `render` component. The `processGrainSrcBufferUpdate()` method is called asynchronously every time the position of the window used for source audio extraction

[2] https://github.com/BelaPlatform/Bela/tree/master/examples/Audio/FFT-phase-vocoder, accessed on 11/05/2020

[3] https://github.com/projectNe10/Ne10, accessed on 11/05/2020

is changed. This was implemented using the AuxiliaryTask component of the Bela framework to prevent underruns when frequently changing the window position. However this does lead to small delays during the update of the source window, as the process is shifted to a separate thread running on a lower priority level than the real time audio thread.

```
void processGrainSrcBufferUpdate(int startIdx){
    // Copy part of sample buffer into FFT input from given start index
    for (int hop = 0; hop < GRAIN_FFT_INTERVAL; hop++){
        int currentStart = hop * FFT_HOP_SIZE;
        for(int n = 0; n < N_FFT; n++) {
            grainSrcTimeDomainIn[n].r = (ne10_float32_t) (gSampleData                  ->
samples[startIdx + currentStart + n] * gWindowBuffer[n]);
            grainSrcTimeDomainIn[n].i = 0;
        }

        // Perform FFT -> indicated by the "0" for the last function parameter
        ne10_fft_c2c_1d_float32_neon (grainSrcFrequencyDomain[hop], grainSrcTimeDomainIn,
 cfg, 0);
    }

    // Update grain source buffer for all playing voices
    for (int i = 0; i < NUM_VOICES; i++){
        if(voiceIndices[i] > NOT_PLAYING)
            voiceObjects[i].updateGrainSrcBuffer(grainSrcFrequencyDomain);
    }
}
```

Listing 1: Method used to update the source buffer

Voice

As discussed above, the Voice component is responsible for masking the windowed source signal. Each instance of the Voice class contains a separate audio buffer, as every voice masks the windowed signal differently, producing differently pitched sounds. To save memory and computing time, each voice contains a reference to the current representation of the windowed signal. That also allows for real time manipulation of the source material, as shown in listing 1.

In addition to synthesis, voices also act as the interface between the main render code and grains. That means that all parameters concerned with the manipulation of grains are accessible through the voice. Furthermore all voices share a reference to the window data used to modify the amplitude envelopes of grains.

Grain

Grains are the fundamental component of the system. They are contained in voices, and share the same audio buffer (per voice). As such, objects of the Grain class do not contain audio information per se, but are defined simply by a starting index in the shared buffer, and a length in samples. This minimises the overhead necessary to access grain information, as all grain-related processes operate on the same buffer object stored by the associated voice.

Window

Four window functions (cf. table 1) were implemented to provide extensive control over the volume envelope of grains. This functionality is encapsulated in the Window component. More specifically, the setLength() method contains the code for the calculation of the four window types. The reason for this is that the window needs to be recalculated every time grain lengths are changed in the user interface. All windows are symmetrical around the centre of the x-axis.

The default window data for grain envelopes is calculated using the Hann window function [9]. It is defined as

$$w(n) = \frac{1}{2}\left[1 - cos\left(\frac{2\pi n}{N}\right)\right] \qquad \textit{Eq. 3}$$

where n is the current sample point and N is the length of the window in samples. The resulting window resembles a bell-like structure in appearance. It strongly attenuates samples close to its start and end points, and gradually rises/falls towards the centre. While it offers a good starting point and is therefore commonly used in granular synthesis, it lacks parameters to modify its structure.

The Tukey window function is an extension of the Hann function that introduces a parameter which specifies a *truncation height.* At this truncation height the window is then cut horizontally, and the last point of the first part is connected to the first point of the second part. To reach the maximum amplitude value of 1 independently of the truncation height, the resulting structure is then stretched vertically until the connecting "line" reaches at a value of 1. The formal definition of the Tukey window function is given by first calculating a factor

$$f(n) = \frac{1}{2t}\left[1 - cos\left(\frac{2\pi n}{N}\right)\right] \qquad \textit{Eq. 4}$$

where t is the truncation height in the range $[0...1]$ and N is the length of the window in samples, and subsequently determining the window value at position n through

$$w(n) = \begin{cases} f(n), & f(n) < 1 \\ 1, & f(n) > 1 \end{cases} \qquad \textit{Eq. 5}$$

By modifying the truncation height, the window can be manipulated to exhibit a steep rise and fall, at the beginning/end of the window, which leads to a more continuous sound if several grains are played simultaneously. However, due to the reduced attenuation, a low value of t may result in clicks, given that the source signal is sufficiently high in amplitude. If t takes the value 1 the Tukey window collapses into the Hann window.

The third window function implements a Gaussian window, which is given by

$$\mathrm{w}(n) = \exp\left[-\frac{1}{2}\left(\frac{n - N/2}{\sigma N/2}\right)^2\right] \qquad \textit{Eq. 6}$$

where σ is the modifier value in the range $[0...1]$ and N is the window length in samples. Large values of σ lead to minimal attenuation of grains passing through, while lower values can be applied to define a narrow pass band around the centre of the window. This leads to a fluttering sound, which can be used to simulate movement in combination with a high number of grains.

The fourth window type is the trapezoidal window, which creates a linear increase of amplitude towards the centre of the window, followed by a symmetrical linear decrease towards the end of the window. It can be modified to steepen/flatten the rate of the increase and decrease of values, which leads to the window reaching a value of 1 before reaching the centre, and values not reaching 1 at all, respectively. For high modifier values the trapezoidal window closely resembles the Tukey window function. Low values however produce a triangular shape, which gives its own distinct sound. To calculate the window it is useful to first determine the fractional representation of the current position in the window

$$p(n) = \frac{n}{N} \qquad \textit{Eq. 7}$$

where N is the length of the window in samples, and use this value to calculate two auxiliary factors

$$f_1(n) = p(n) * s \qquad \textit{Eq. 8}$$

and

$$f_2(n) = -1 * s * \left(p(n) - \frac{s-1}{s}\right) + 1 \qquad \textit{Eq. 9}$$

where s is the modifier value in the range $[0...10]$. Using those values the final window function is given by

$$w(n) = \begin{cases} \begin{cases} f_1, & f_1 < 1 \\ 1, & f_1 \geq 1 \end{cases}, & p(n) < 0.5 \\ \begin{cases} f_2, & f_2 < 1 \\ 1, & f_2 \geq 1 \end{cases}, & p(n) \geq 0.5 \end{cases} \qquad \textit{Eq. 10}$$

Filters

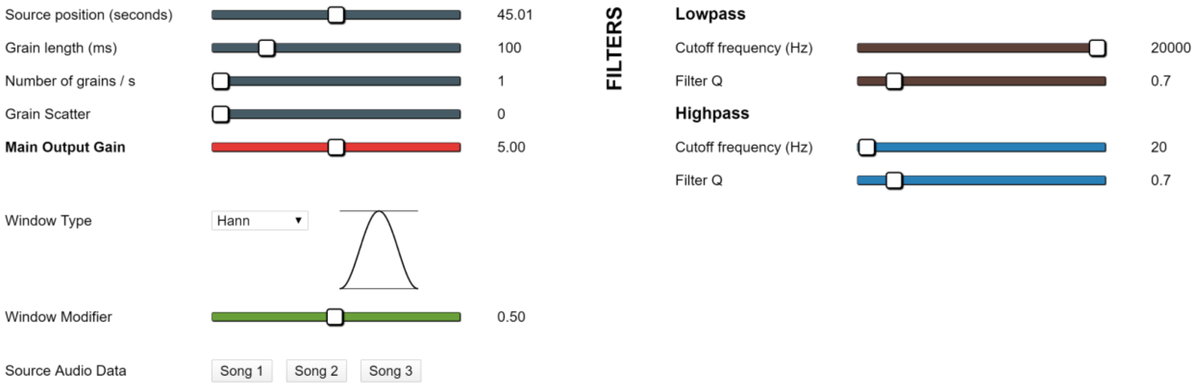
Two digital filters were implemented to allow for attenuation of the high-/low-end of the output sound. The first filter is a fourth-order lowpass filter that was adapted from the system designed for the first assignment of Music and Audio Programming. It consists of two sequential biquad filters. The second filter is a second-order highpass filter. Here the calculation of the filter coefficients was adapted from the JUCE framework implementation[4].

Graphical User Interface

While not directly related to the processing of sounds, the user interface is still an integral part of the system, as it provides controls for all its parameters as well as mechanisms for visual feedback about the current state. The interface was created using the *p5 js* library and

[4] https://github.com/juce-framework/JUCE/blob/master/modules/juce_dsp/processors/juce_IIRFilter.h, accessed on 12/05/2020

connected to Bela using the provided system functions. The code describing its functionality is included in the `main.html` and `sketch.js` files. A screenshot of the GUI is shown in figure 2.



Source position (seconds) 45.01
Grain length (ms) 100
Number of grains / s 1
Grain Scatter 0
Main Output Gain 5.00
Window Type Hann
Window Modifier 0.50

FILTERS
Lowpass
Cutoff frequency (Hz) 20000
Filter Q 0.7
Highpass
Cutoff frequency (Hz) 20
Filter Q 0.7

Figure 2: Graphical user interface

The interface is divided into three subsections. The main control elements are located at the top left. Those parameters directly affect the composition and therefore the sound of grains. The *Source position* slider controls the starting position of the extraction window in the loaded sample. The *Grain length* slider controls the length of all grains across all voices. The number of grains that are triggered per second by *each voice* can be defined with the third slider. The *Grain Scatter* slider introduces pseudorandom modifiers to the starting positions and lengths of grains. The resulting sounds are less concise, and more distributed across the extraction window. This can be exploited to create sonic textures. The *Main Output Gain* slider applies a gain value to the output signal. It is configured to span values in the range of $[0...20]$, as a significant part of the original signal information is discarded in the masking process prior to grain selection, which generally leads to more quiet grains.

Below the main control section are additional controls that do not directly affect the timbral quality of the sound. The *Window Type* selection offers the four types of window functions described above. The *Window Modifier* slider can be used to further configure the Tukey, Gaussian and trapezoidal window types. Changes in the window structure are communicated to Bela, which returns the updated window data to the user interface. A visualisation of the current window function is created every time the window parameters change. This allows for a crude estimation of the window characteristics and consequently the way the window will affect the sound. The *Source Audio Data* row provides three buttons to switch between source audio material. The three songs are pre-loaded in the `main.cpp` script and stored in three audio buffers. Switching between source material is possible during playback.

The *Filters* section provides controls for the two filters. Their cutoff frequencies and q-values can be changed here.

NB: In the current implementation there exists a bug in the connection between Bela and the user interface. Sometimes sending a single change event (e.g. clicking at the target position of a slider to change its value instead of dragging it) does not have an effect on the system. If this happens it is recommended to drag the slider slightly to the left/right. Similarly to make sure that the source buffer is changed upon clicking one of the three source audio data buttons, the respective button should be pressed twice.

Evaluation

To demonstrate the capabilities of the system, several audio files were created from the three available input files included in the project ("*betti.wav*", "*nicefornothing.wav*" and "*jazzo.wav*"). The audio files were recorded using an external speaker connected to Bela. The speaker was placed directly in front of a Rode NT-1A microphone, which routed the sound into a digital audio workstation. This method of recording was chosen due to the low input levels observed when routing Bela directly into an audio interface (no 5V power adapter and respective TRS cables were available). All produced output audio files are included in the "*output*" folder of the project submission bundle.

The first example is designed to demonstrate the system's range of timbral capabilities from a single source file. Using the third audio file as source, four sections were recorded, containing 8 repetitions of a middle C each. The recordings were concatenated into one audio file, which was subjected to analysis using the Sonic Visualiser software [10]. Figure 3 shows the log-magnitude spectrogram representation of the concatenated audio data. Changes to the source position can be observed at the 4, 8 and 12 second marks respectively. Due to the change of the source material the amplitudes of the fundamental frequencies and their overtones change significantly, which is perceived as a change in the timbre of the sound. It should be noted that the *composition* of overtones is unchanged. This is due to the system's static masking functionality, which uses the fundamental frequency and 20 overtones to create the frequency mask.

The second example illustrates the effect of different window functions on a single grain with a length of 250ms. Figure 4 shows the spectrogram of a concatenated recording with different window functions applied. The window functions and values per note event are given in table 2:

Note event #	Window function	Modifier value
1	Hann	-
2	Tukey	0.5
3	Tukey	0
4	Gaussian	0.1
5	Gaussian	0.2
6	Gaussian	0.5
7	Gaussian	1.0
8	Trapezoidal	10.0
9	Trapezoidal	3.0

Table 2: Window functions and modifier values for example 2

As reflected in the spectrogram, the biggest difference in output characteristics is observed when using the Gaussian window function with relatively low modifier values. The narrow band of the resulting window audibly shortens the perceived grain duration. The effects of the other window functions are more subtle and can be heard in the output file "*nicefornothing_window.wav*". A second audio file containing repeated *chords* with varied window function parameters was recorded as well ("*nicefornothing_window_addon.wav*"). The development of window function parameters in the second file is similar to the first one.

A final recording (“*betti_variable_parameters.wav*”) was made to show the interplay between the system’s parameters and the resulting sound. In the beginning, three voices are triggered to produce a chord, with grain lengths of 100ms. First the grains are gradually stretched, after which the frequency of the grains, i.e. the number of grains per voice per second, is increased. Then the *scatter* parameter is introduced, which leads to a pseudorandom distribution of grain start times and lengths. Finally the lowpass filter’s cutoff frequency is lowered to reduce some of the high frequency content, followed by an increase of the highpass filter’s cutoff frequency, which cuts of the low end, producing the bandpass-like sound in the end.

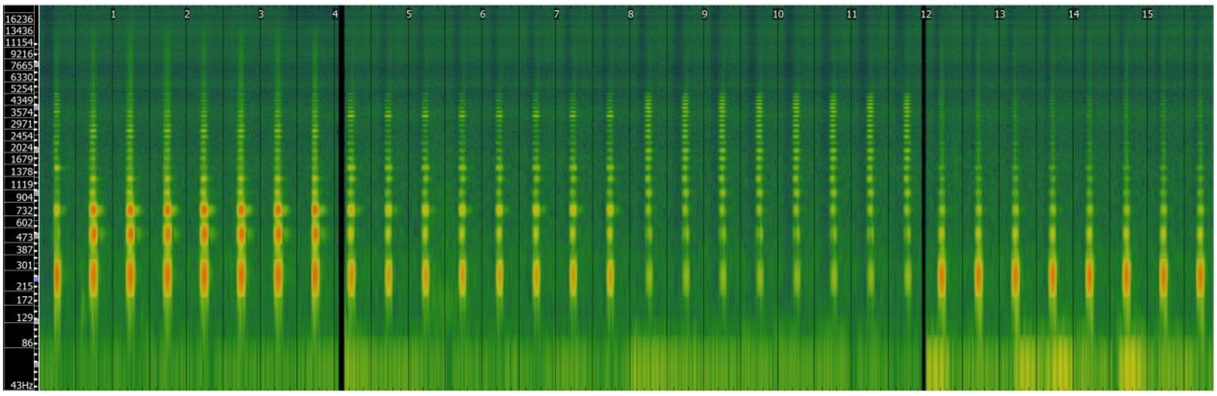


Figure 3: Spectrogram of the output of example 1 (“jazzo_one_note_source_pos_varied.wav”)

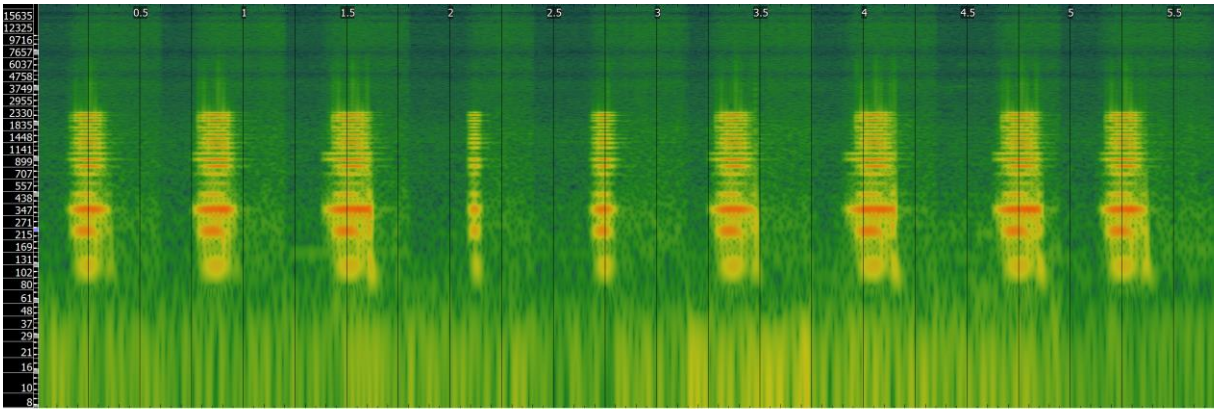


Figure 4: Spectrogram of the output of example 2 ("nicefornothing_window.wav")

Limitations

Arguably the most striking limitation of the system in its current implementation is the frequency resolution of the short-term Fourier transform. The current configuration of 4096 samples for the window size and 1024 samples for the hop size was found empirically to produce the best results while avoiding underruns. Assuming a sampling rate of 44100Hz, this provides a frequency resolution of $\frac{22050}{2049} = 10.76$ Hz per FFT bin. Table 3 shows the resulting inaccuracies for several notes and their corresponding frequencies, measured in a digital audio workstation using a Virtual Studio Technology plugin[5] for tuning. The error is given in cents. It was found that differences in octaves lead to no significant differences in recognition rate,

[5] https://www.gvst.co.uk/gtune.htm, accessed on 12/05/2020

contradicting the initial assumption that error rates of frequencies decrease as their pitches increase. The same effect was found with a different tuning plugin[6]. One explanation might be that the underlying algorithms of the tuning plugins combine the accuracy scores of fundamental frequencies and their overtones. Due to the use of 20 overtones in the current implementation, and their naturally higher *relative* accuracy, this might explain the results. However, below the note F2 (87.31Hz), recognition failed. Thus we assume that this constitutes the lowest possible note giving the correct pitch in the current implementation.

Note	Fundamental Frequency	Error (cents)
C2	65.41	- (Not recognised)
C3	130.81	-15
C4	261.63	-1.2
C5	523.25	-0.9
C6	1046.50	-1.7

Table 3: Inaccuracies in pitch recognition per octave

One further limitation of the current implementation is that the graphical user interface must be opened once via the Bela web IDE after the project is started in order to allow for the output of sound. The reason for this is that some parameters are currently initialised through the user interface. Furthermore, for the graphical user interface to be displayed correctly, the project must be located in the "*projects*" folder on Bela and the project (i.e. folder) name must be "*pitch-aware-granular-synth*". This is due to the use of a custom `main.html` file used to style the GUI.

Regarding sound, only mono input files for source material are supported at the moment. Finally, the current number of voices is limited to 10, and the number of grains per voice is limited to 30.

Conclusion

We presented an algorithm for deterministic and pitch-accurate granular synthesis of sounds independent of the source material. We discussed its functionality, evaluated its performance and demonstrated its feasibility both as a digital musical instrument and a device for sound design. Future work could refine the system's frequency resolution in order to increase the perceived accuracy of individual notes. In terms of musicality, efforts could be made to incorporate additional MIDI control parameters, such as pitch-bending and velocity sensitivity to enhance the system's expressive capabilities. Additionally, different methods of calculating the constitution of the set of overtones could be implemented to offer an additional layer of timbral variation.

[6] https://www.ableton.com/en/manual/live-audio-effect-reference/#22-35-tuner, accessed on 12/05/2020

References

[1] Gabor, D. 1946. Theory of communication. Part 3: Frequency compression and expansion. *Journal of the Institution of Electrical Engineers - Part III: Radio and Communication Engineering, vol. 93, no. 26, pp. 445-457.*

[2] Xenakis, I., 1992. Formalized music: thought and mathematics in composition (No. 6). Pendragon Press.

[3] De Poli, G., Piccialli, A. and Roads, C. eds., 1991. *Representations of musical signals.* MIT press.

[4] Truax, B., 1988. Real-time granular synthesis with a digital signal processor. *Computer Music Journal, 12*(2), pp.14-26.

[5] Miranda, E.R., 1995. Granular synthesis of sounds by means of a cellular automaton. *Leonardo, 28*(4), pp.297-300.

[6] Keller, D. and Truax, B., 1998, October. Ecologically-based Granular Synthesis. In *ICMC.*

[7] Fröjd, M. and Horner, A., 2009. Sound texture synthesis using an overlap–add/granular synthesis approach. *Journal of the Audio Engineering Society, 57*(1/2), pp.29-37.

[8] Allen, J., 1977. Short term spectral analysis, synthesis, and modification by discrete Fourier transform. *IEEE Transactions on Acoustics, Speech, and Signal Processing, 25*(3), pp.235-238.

[9] Blackman, R.B. and Tukey, J.W., 1959. Particular pairs of windows. *The measurement of power spectra, from the point of view of communications engineering*, pp.98-99.

[10] Cannam, C., Landone, C., and Sandler, M., 2010. Sonic Visualiser: An Open Source Application for Viewing, Analysing, and Annotating Music Audio Files. *Proceedings of the ACM Multimedia 2010 International Conference.*

[11] Fechner, G.T., 1966 [First published 1860]. Howes, D. H., Boring, E. G. (eds.). *Elements of psychophysics [Elemente der Psychophysik]*. Volume 1.

Appendix

- A video explaining and demonstrating the system is available on YouTube: https://youtu.be/rtKI67ztNYo
- The spectrogram image in the signal chain workflow was taken from https://commons.wikimedia.org/wiki/File:Spectrogram_of_violin.png, accessed on 11/05/2020
- The waveform representation in the signal chain workflow was taken from https://www.needpix.com/photo/776825/sound-wave-waveform-aural-audio-sonic-ear-hearing-music, accessed on 11/05/2020
- The code for running this project is publicly available on GitHub: https://github.com/maxgraf96/pitch-aware-granular-synth