Granular Synthesis for Instrument Makers

Daniël Kamp
HKU University of the Arts Utrecht
Department of Music and Technology
daniel.kamp@student.hku.nl

***Abstract*—This writing sheds light onto various forms of granular synthesis. It analyses some existing applications, explains the inner workings of such systems, and proposes methods of how one may implement the technique themselves.**

Introduction

Granular synthesis is a versatile form of sample-based (digital) sound synthesis. It uses short slices of recorded audio to create new, rich sounds. These slices can be layered, stretched, spread, and in other ways processed to create vast soundscapes with relative ease.
This writing aims to demystify the technical aspects of granular synthesis, and provide a comprehensive insight for instrument makers who want to make use of the technique.

I. What is granular synthesis?

This first part tells a brief history of granular synthesis, explains the components of a system implementing this technique, and lists some reference instruments.

A. A brief history

Granular synthesis as it's known today is a form of digital sound synthesis, which is based on the processing of small slices of sampled audio known as "grains". The underlying view on sound perception was first proposed by Hungarian-British physicist Dennis Gabor in an article titled *Acoustical Quanta and the Theory of Hearing* (Gabor, 1947). In this writing, Gabor theorized an approach to sound similar to concepts found in quantum mechanics.
Building on this theory, in the late 1950s, Greek composer Iannis Xenakis started experimenting with the technique, stitching together many tiny slices of tape to create new sounds (Robindoré, 1996). Inspired by Xenakis during a 1972 workshop taught by the former, composer Curtis Roads eventually created the first computer software implementing this synthesis paradigm in 1974 (Opie, 2003). His code produced a 30-second long piece titled Klang-1, using three parameters: envelope, duration, and density. This software, however, did not operate in real-time, and producing this piece of music took multiple days. The first real-time implementation of the principle came from Canadian composer Barry Truax, who developed GSX (1986) and GSAMX (1987) (Truax, 1988).
Since then, granular synthesis has established itself as a pillar of modern digital sound generation. More approachable implementations of the technique, like Robert Henke's Granulator (2011), made it possible for anyone with a computer to explore its vast new possibilities in sound. Today, dozens of granular synthesizers are available, each offering a different approach to a 75-year-old concept.

B. General system overview

This section describes the general anatomy of a granular synthesizer, and gives a broad overview of the building blocks that make up such a system.

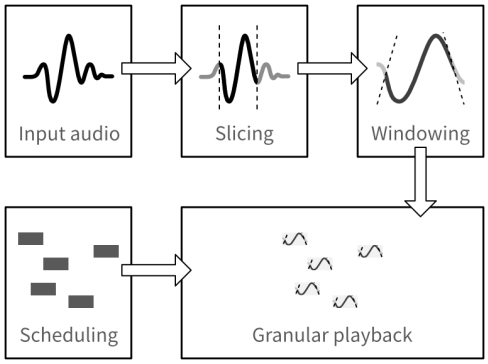

Fig. 1. A simplified overview of a granular synthesis system.

1) Sound source: As its input, a granular system takes a sound stream. This stream can consist of sampled audio, live microphone input, or generated audio (such as sine waves). The content of this source material has an effect on the sounds one can generate: when using a transient-rich source, for example a drum sample, the resulting audio will usually have a substantial amount of content in the higher regions of the frequency spectrum. Sounds with a more periodic envelope are more suitable for generating melodic content, since they usually result in a more mellow output sound.

2) Models: There are four distinguishable models of granular synthesis: screen, cloud, stream, and spray (Roads, 2012). The first of these, screen, was described by Xenakis and would run a series of time-frequency plots at a fixed time rate (Xenakis, 1963). In layman's terms, this would be comparable to a flip book animation, used for sound instead.
The second model, clouds, generates a cloud of sound in which the individual grains are not bound to predefined parameters, but instead spread across specified ranges in a stochastic manner. This creates rich, unpredictable textures that are often used on a macro level within compositions.
The more regular counterpart of clouds is streams, in which

grains are distributed in a deterministic process. In this model, the density (number of grains per second) determines a regular interval at which grains are played, creating a sort of rhythmic texture. This rhythm can be tweaked by modifying the grain density, through which one can create tone-like sounds.
A model that is used in modern digital granular synthesis is spray. This implementation makes it possible for users to draw the grains on a plot of frequency over time, which makes for an intuitive interaction model.

3) Windowing and envelope: Since the start position of grains is rather arbitrary, there is no telling what the amplitude of the first sample in the selection will be. Therefore, it's good practice, as it is in any digital audio application, to apply an envelope that trims off any rough edges that may exist in the source material. This envelope can be modified to a user's own insight through the parameters *shape* and *length*.
The shape of an envelope refers to its mathematical function. In this regard, there are four commonly used shapes: linear, square root, exponential, and Gaussian. The latter was proposed by Dennis Gabor in his original theory on granular sound, and is still commonly used. More recently, due to developments in the field of processing power in computers over the past fifty-or-so years, it has become possible and commonplace to use other envelope shapes as well. Figure 2 shows the four functions as a plot of amplitude over time.

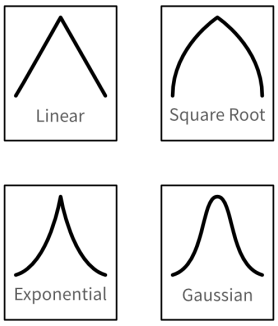

Fig. 2. Four envelope shapes.

The length of an envelope simply describes the time it takes for the signal to reach its full amplitude. By varying this number, one can change the character of the resulting sound. If a long *attack* (rising edge of the envelope) is used, the resulting audio will sound more mellow compared to when a short attack time is applied.

4) Duration and density: Density, as mentioned previously, describes the number of grains played in one (1) second. Variations in this domain can roughly be split into two categories: *deterministic* and *stochastic*.
In a deterministic distribution, grains are played back with even spacing between them. This means that the resulting sound is of a regular nature. For example, this method would produce audio in which a grain is played every 100 ms.
Stochastic distribution, on the other hand, adds a realm of uncertainty to the sound. When this distribution is applied, the beginning and length of each grain are randomized. This means that the resulting sound is no longer predictable and/or regular in nature. Using this method, one can create vast clouds of sound that contain no structure or repetition.

C. Existing instruments

This section lists a few instruments and effects that use granular synthesis.
Author's note: I selected open-source instruments of which the source code is freely available to the public. This way, I can give an in-depth comparison of the inner workings of these instruments. I'll use these and their concrete implementations as references throughout the rest of my research described in this writing.

1) Mutable Instruments' Clouds: Clouds is a Eurorack module developed and produced by Émilie Gillet (Mutable Instruments). Both the hardware files (schematics, CAD-files) and firmware source code are open-source and publicly available (Gillet, 2014).
As its source sound, Clouds uses incoming stereo audio compliant with the Eurorack standard.
Looking at the source files of Clouds, one can dissect the process used to "seed" grains. First, a maximum number of grains is "hard-coded" into the software. The program will create new ones until this limit is reached. Clouds offers both deterministic and stochastic seeding modes, which can be selected by the user. In deterministic mode, one controls the time between seeds in a regular manner. In stochastic mode, the user input controls the likelihood with which grains will be created.
This implementation uses so-called Look Up Tables (LUTs) to store data for its envelopes and other resource-intensive processes. These tables contain pre-calculated numbers, so that the program doesn't need to calculate these in real-time. This makes it suitable for low-power devices (in case of Clouds, this is an STM32 micro-controller). The next chapter contains examples of, and a comparison between this implementation and one that does calculate its data in real-time.

2) Argotlunar: Argotlunar is a real-time granular VST and AudioUnit plugin created by Michael Ourednik. The source code of this software is open-source and available via GitHub (Ourednik, 2021). This granular effect is implemented using the JUCE framework.

Since Argotlunar is designed to run as a plug-in within a Digital Audio Workstation, it will usually have more computing resources available than the previous example. This is apparent in its code, as liberties have been taken that would probably be avoided on embedded systems. For example, a lot of the variables that are needed during sound processing are generated at runtime. In low-power systems, it's common practice to avoid this by calculating the data beforehand and compiling them into the program.

II. How does granular synthesis work?

This part dives deeper into the inner workings of a granular system. Starting with a brief explanation of core audiological concepts that define how we experience sound on a short time scale, it then proceeds to explain concrete implementations of a granular synthesizer in C++. These examples are then compared through statistics about their performance, so a recommendation can be made based on system efficiency.

A. Audiological concepts

The granular synthesis technique relies heavily on the psycho-acoustical concept of *forward masking* (Bates, 2004). This "limitation" of human perception causes sound impulses at quick succession to blend together and be perceived as a continuous tone. In numbers, a threshold for this interval between two impulses can be established at around 10ms (Gabor, 1947), which corresponds to a frequency of 100Hz. However, if succeeding impulses are played at half the intensity of that first impulse, the time between sounds can be as long as 21ms. In other words, if a short sound is played every 10-21ms, a human listener will perceive this as a continuous sound, and may even discern a (musical) tone.

B. Runtime constraints

The information above, however, only applies in part to the digital world. Instead of generating sound in real-time, a computer uses an audio engine that renders its output in blocks. This system is capable of rendering a number of cycles at once. These are then scheduled by a separate process, allowing sound generation and playback to operate independently. This gives the rendering process more time to perform its calculations.

One such audio engine is the JACK Audio Connection Kit, a cross-platform toolkit for building and running audio applications. Using this program as an example, it's possible to calculate the constraints within which an implementation needs to operate.
In the author's setup, JACK has a default sample rate of 44.1kHz, and operates with a block size of 1024 frames per period. With this information, the time per block can be calculated as follows:

$$rendering\ time = \frac{block\ size}{sample\ rate}$$

This returns a rendering interval of around 23ms per block. Dividing it by the number of samples that need to be rendered during this time, the rendering time per cycle comes down to about 22μs. For simplicity, one may assume that this entire rendering time is available to the granular synthesis process.

C. Implementations

In this section, implementations and methods are proposed for specific parts of a granular system. The next section will take execution metrics from these example implementations, and validate them by the requirements defined previously.

1) Scheduling algorithms: This section gives four examples of grain scheduling algorithms, using two different approaches. These algorithms determine when grains are played, and thus heavily influence the resulting sound's character.

Deterministic approach

For a deterministic method, a user directly or indirectly controls the sound's *density*, using parameters like *grain length* and *grain interval*. Playing back grains with a regular interval creates a predictable and periodic output sound, which can even be used in a melodic way. The following code example implements a deterministic approach for a single stream of grains.

```
int interval = grain_size / density;
for (int i = 0; i < (block_size - interval); i +=
  interval) {
  grains.push_back(Grain(i));
}
```

Listing 1. A deterministic method of grain "seeding"

Stochastic implementation 1: random start position

To add variation to a sound, one might want to depart from a periodic triggering algorithm like the one described above. The cloud model, as mentioned in section 1.B, exchanges the regular interval for random chance.
This first implementation uses randomness to determine the starting position of each grain. To generate its random numbers, it uses C++'s built-in `std::rand()` function.

```
int max_grains = floor(MAX_GRAINS *
  spawn_likelihood);
for(int i = 0; i < max_grains; i++) {
  float v = (float) rand() / (float) RAND_MAX;
  grains.push_back(Grain((int) (block_size * v)));
}
```

Listing 2. A stochastic method of grain seeding: random start position

Stochastic implementation 2: random chance

This second stochastic method loops through the rendering block and uses randomness to determine whether a grain is to be scheduled on the current position. It includes a minimum offset between grains, so that a high percentage of the block is guaranteed to be covered. Additionally, it swaps the built-in `std::rand()` function for the `<random>` library that is included with the C++ Standard Library.

```
random_device rd;
mt19937 gen(rd());
uniform_real_distribution<> dist(0, 1);
int wait = 0;

for(int i = 0; i < block_size; i++) {
  if(wait == 0) {
    v = dist(gen);
    seed = v < spawn_likelihood;
    if(max_grains > 0 && seed) {
      grains.push_back(Grain(i));
      max_grains--;
      wait = minimum_interval;
    }
  } else {
    wait--;
  }
}
```

Listing 3. A stochastic method of grain seeding, using `<random>`

Combined approach: Divide-down

To add rhythm to the chaos of using a stochastic approach, one might add a deterministic element. Figure 3 shows how a combination of these two implementations might look. In this example, each grain has a random chance to be seeded.

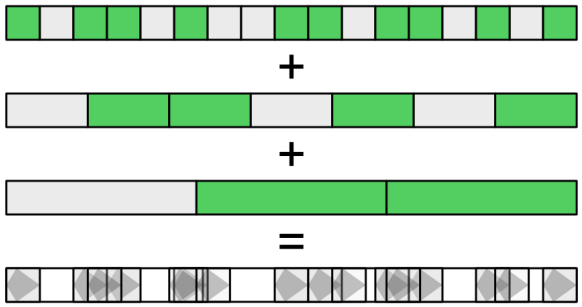

Fig. 3. A divide-down algorithm visualized.

During each loop, the full length of the rendering block is divided by the remaining number of grains. By layering these divisions, one can create a cloud of unpredictable sound, while still retaining a certain structure.

The following code combines the examples from **Deterministic approach** and **Stochastic approach 2**.

```
int max_grains = 64;
int number_of_grains = 0;
int interval = (float) block_size / (float)
  max_grains;

while(max_grains > 1) {
  interval = (float) block_size / (float)
  max_grains;

  if(interval > minimum_interval) {
    for (int i = 0; i < (block_size - interval); i
   += interval) {
      if(max_grains > 0) {
        v = dist(gen);
        seed = v > spawn_likelihood;
        if (seed) {
          grains.push_back(Grain(i));
          max_grains--;
        }
      }
    }
  } else {
    max_grains = 0;
  }
}
```

Listing 4. An implementation of a divide-down algorithm

2) Windowing algorithms and approaches: This section compares three approaches for implementing an envelope generator. For simplicity, each method renders a quadratic parabola envelope shape with a duration of 4096 samples.

First implementation: Pre-rendering

This method calculates a new lookup table when any values (such as envelope or grain length) are changed. This way, the program can simply look up and apply the right value during the rendering process, without needing additional calculations.

```
#define ENV_ROUNDS 4096

float envelope[ENV_ROUNDS];
float intermediate_;

// Initialization, runs once
for(int i = 0; i < ENV_ROUNDS; i++) {
  intermediate_ = ENV_SHAPE * (float)(-1.0 * i * i
   + ENV_ROUNDS * i);
  envelope[i] = (intermediate_ >= 1.0) * 1.0 + (
  intermediate_ < 1.0) * intermediate_;
}

// Rendering loop, runs while active
for(int i = 0; i < ENV_ROUNDS; i++) {
  test1[i] = sample[i] * envelope[i];
}
```

Listing 5. Envelope application using a pre-rendering algorithm

Second implementation: Look-Up Table

In this next example, the program first scales its position in relation to the generated look-up table. Using this scaled position, it looks up the nearest corresponding value. It then uses a low pass filter to interpolate between the current and previous value.

The following formula was used to generate the data for the look-up table:

$$y_n(x_n) = \frac{0.001 * ((\text{-}x_n^2) + \textit{grain size} * x_n)}{\textit{grain size}}$$

$$f_n(y_n) = \frac{3}{2} * \begin{cases} \frac{y_n^3}{3} & y_n < 1.0 \\ \frac{2}{3} & y_n \geq 1.0 \end{cases}$$

```
// Initialization, runs once
float ENVELOPE_LUT[6000] = {0, 0.00149975,
  0.002998996, 0.004497737, ...};

auto scale_factor = (sizeof(ENVELOPE_LUT) / sizeof
  (ENVELOPE_LUT[0])) / ENV_ROUNDS;
float previous = ENVELOPE_LUT[0];
int ptr = 0;
float tmp = 0.0;

// Rendering loop, runs while active
for(int i = 0; i < ENV_ROUNDS; i++) {
  ptr = floor(scale_factor * i);

  tmp = 0.5 * (envelope[ptr] + previous);
  test2[i] = tmp * sample[i];

  previous = envelope[ptr];
}
```

Listing 6. Envelope application using a look-up table

Third implementation: Real-Time Rendering

This last example is taken directly from the Argotlunar source code. It calculates a step and slope value, and uses these to increment the envelope's multiplication variable.

```
// Code by M. Ourednik
// Initialization, runs once
float env_amp = 0.0f;
float d = 1.0f / ENV_ROUNDS;
float d2 = d * d;
float slope = 4.0f * 0.8 * (d - d2);
float curve = -8.0f * 0.8 * d2;

// Rendering loop, runs while active
for (int i = 0; i < ENV_ROUNDS; i++) {
  env_amp += slope;
  slope += curve;
  test3[i] = sample[i] * env_amp;
}
```

Listing 7. Envelope application using real-time rendering

D. *Efficiency*

For this section, the implementations from above were ran multiple times, to gain an accurate result. The execution time of each loop iteration has been recorded, along with the initialization time. Note that these benchmarks were ran on a 2019 MacBook Pro (Intel Core i7 6-core processor, 16GB RAM).

1) Scheduling algorithms: The first results describe the performance of the scheduling algorithms proposed earlier. These implementations are compared not by execution time necessarily, but more so by adherence to a user's intention. For example, if a user turns the density knob, the resulting sound should reflect the desired changes.

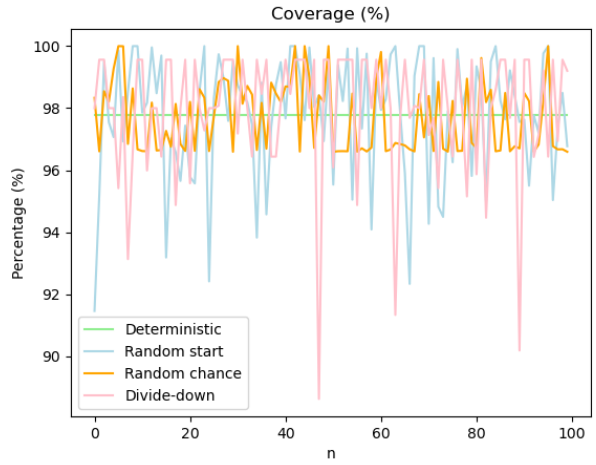

Fig. 4. Percentage of the rendering block covered by each algorithm, measured 100 times.

This coverage analysis shows that all methods spread the grains evenly over the entire block. A deterministic approach always ensures the same coverage, which is to be expected when all parameters stay the same. Out of the stochastic methods, the divide-down algorithm shows the most variation in density between blocks. The other two stochastic algorithms perform rather similar.

In the next test, the linearity between user input and the density of the output were measured. This was done by increasing the input value with $\frac{1}{100}$ over 100 total measurements.

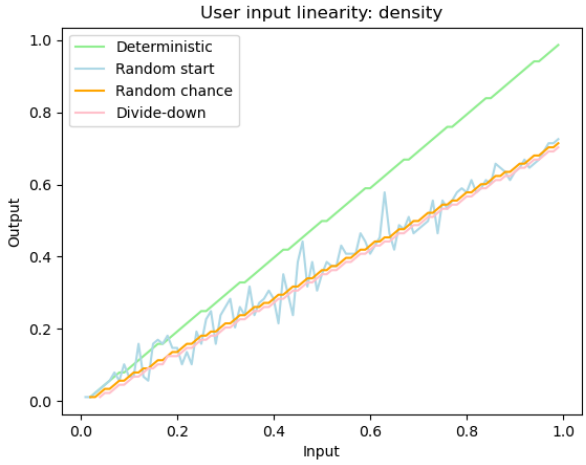

Fig. 5. Correlation between user input and system output. Measured on the interval $input = [0, 1]$.

These results show a direct correlation between the user's input and the system's output density. As to what causes the difference in slope: this is currently unknown, and requires further investigation. The author believes it to be a result of the relation between grain size and interval.

In this next analysis, the execution time per cycle was measured for each of the four different implementations.

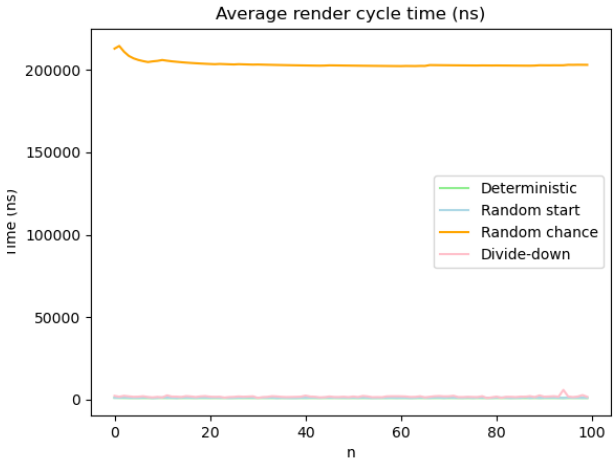

Fig. 6. Average time per scheduling rendering cycle, measured 100 times.

The obvious takeaway from this data is that the *random chance* algorithm takes extremely long to complete its rendering. In the graph, this makes it impossible to analyze the other data. When leaving out this method, one can get a better picture of the performance of these other implementations. This detailed view is pictured in Figure 7.

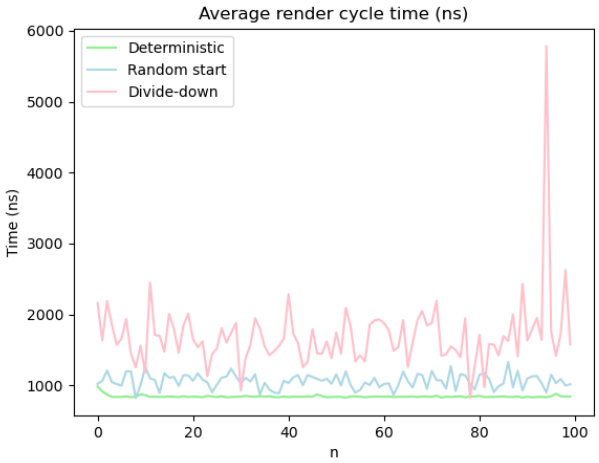

Fig. 7. Average time per scheduling rendering cycle, measured 100 times, without *random chance*.

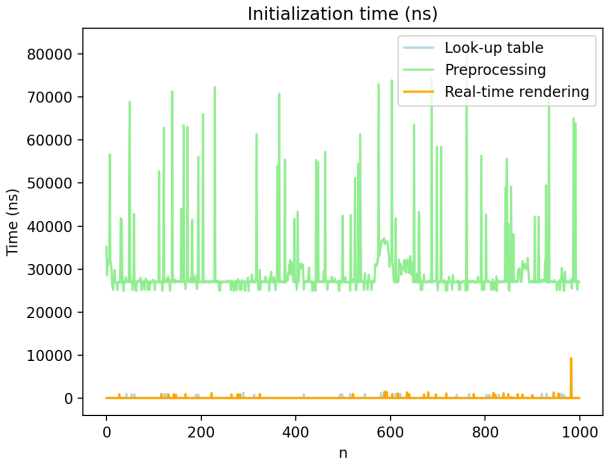

Fig. 8. Initialization time, measured 1000 times.

This graph shows comparable performance for the *random start* and *divide down* approaches. They both have irregular execution time, which is to be expected when using random variables. The latter, however, takes slightly longer to process, likely due to its reliance on recursion. Still, the two algorithms perform similarly enough to be considered interchangeable.
Next, the *deterministic* approach stands out due to its extremely short processing time. This is likely due to the fact that this method uses a far shorter loop than all other implementations. Instead of stopping at each sample in the block, it increments its position by a calculated interval. Because of this implementation, the loop has less code to execute, and thus takes significantly less time.

Keeping in mind the maximum rendering time of 23μs, as calculated previously, it's clear that the *random chance* algorithm, with an average rendering time of 203μs, does not meet the criteria. Comparing the other three outcomes, with averages of 843ns, 1.1μs, and 1.7μs respectively, one would likely choose to implement a method with a low rendering time. However, it must be noted that all three algorithms have varying use-cases and render different results. While the *deterministic* method might have a short execution time, its output may be too repetitive for some applications. Nevertheless, as a product solely of these statistics, it's unlikely that any of the three remaining approaches would offer insufficient performance for most applications.

2) Envelopes: Next, the benchmark results of the envelope implementations show some interesting details. Before looking at the runtime statistics, let's analyze the initialization times of each algorithm. Figure 8 shows this data, measured in nanoseconds. Looking at this graph, one will notice that the pre-processing implementation takes significantly longer to initialize than its counterparts. This can be attributed to the way this method renders its data all at once during this initialization phase. However, this probably won't affect the program's execution, as the initialization code is only ran sporadically and does not need to fit within the 23μs rendering time constraint.

The second graph, however, paints a different picture. Comparing the results in Figure 9, both the pre-processing and real-time algorithm have an average render cycle time of around 42ns, while the look-up table method takes over 10ns longer. This puts especially the *real-time rendering* method in a very good position, as it combines both a low average rendering time with an already fast initialization.

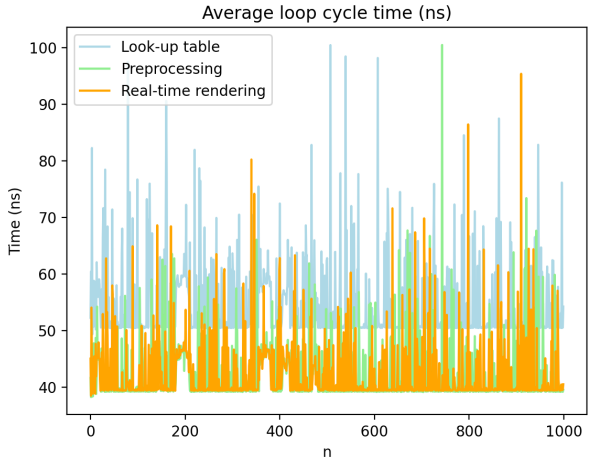

Fig. 9. Average time per rendering cycle, measured 1000 times.

Finally, it may prove insightful to add some times together in order to provide a broad estimate of what rendering times for a complete granular implementation could look like. This does not reflect any additional processing such as filtering and recording, so one should take these results with a *grain* of salt. An example implementation using a *divide-down* algorithm to schedule its grains and then applies an envelope using *real-time rendering* will take approximately 450ns to complete. This accounts for about 2% of the total available time. Another combination would add *preprocessed* envelopes to a *deterministic* scheduling approach, adding up to around 1.1μs and consuming about 4.7% of the rendering cycle. As can be concluded from these two examples, a granular process

like this should generally not add a big overhead to an audio rendering process.

Conclusions

Over the past 75 years, granular synthesis has developed itself from cutting edge physics theory to commonplace tool in a musician's workshop. Thanks to the findings of scientists like Dennis Gabor and the work of musicians like Curtis Roads, instrument makers have yet another flavor to liven up their creations. When put to the test, this technique has proven itself rather versatile in both its applications and performance. Its existing implementations range from VST instruments to hardware effects, and the number of up-and-coming producers and composers using the technology is growing by the day.
This writing only covered a small share of many possible approaches to implementing elements of this technique, and therefore should not be considered a definitive list of available methods. It can, however, serve as the basis for new ideas and inspire completely new applications and implementations. The research has been done, the code has been written, and now it's time to play.

References

Bates, E. (2004). *Composing, perceiving and developing real-time granulation in surround sound* (Master's thesis). Trinity College Dublin.

Gabor, D. (1947). Acoustical quanta and the theory of hearing. *Nature*, *159*, 591–594.

Gillet, É. (2014). *Clouds* [Source code]. Retrieved May 6, 2022, from https://github.com/pichenettes/eurorack/tree/master/clouds

Opie, T. (2003). *Creation of a real-time granular synthesis instrument for live performance* (Master's thesis). Queensland University of Technology.

Ourednik, M. (2021). *Argotlunar* [Source code]. Retrieved May 3, 2022, from https://github.com/mourednik/argotlunar

Roads, C. (2012). From grains to forms. In M. Solomos (Ed.), *Proceedings of the international symposium "Xenakis. La musique électroacoustique"*.

Robindoré, B. (1996). Eskhaté Ereuna: Extending the limits of musical thought - comments on and by Iannis Xenakis. *Computer Music Journal*, *20*, 11–16.

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

Xenakis, I. (1963). *Musiques formelles: Nouveaux principes formels de composition musicale.* Editions Richard-Masse.

All graphics in this writing were created by the author, unless otherwise noted.