Euclidean Algorithm for auto-generative patterns in a Supercollider application

Project of Computer Music course

Students:
De Bari Mauro Giuseppe Matr.899371
Albertini Davide Matr.883347
October 18, 2018

Contents

1 Introduction

Euclidean rhythms are a large class of rhythms which can be generated through the Euclidean Algorithm. Making its first appearence in Euclid's Element, it's one of the oldest algorithm, and it was used to compute the greatest common divisor of two given integers. It was Godfried Toussaint to discover, in 2005, that the Euclidean algorithm's structure may be used to generate the aforementioned class of rhythms. Moreover, he found that those patterns are often present traditional and world music[1]. He noticed that the main characteristic of those rhythms is that their onset pattern are distribuited as evenly as possible. Since then, Euclidean rhythm generators started to spread in the music production world, since with very few parameters one can generate very complex patterns and polyrhythms. A lot of eurorack synthesizer modules have been developed: Mutable Instruments Yarns[2], vpme.de Euclidean Circles[3] and 2HP Euclid[4], just to name a few. This paper will present a software implementation of a drum machine capable of generating Euclidean rhythms, using Supercollider programming language.

2 The algorithm

The purpose of the researchers in the past year was to find a easy way to generate all the possible rhythms present in music, in order to extend the dictionary of Automatic Music Composition[5]. Before proceeding with the implementation of the drum machine, it's necessary to dwelve into the algorithm of Euclid and explain what it has to do with rhythm generation.
In order to compute the greatest common divisor between two integers, the greek mathematician proposed this solution: the smaller number is repeatedly subtracted from the greater until the greater is zero or becomes smaller than the smaller, in which case it is called the remainder. This remainder is then repeatedly subtracted from the smaller number to obtain a new remainder. This process is continued until the remainder is zero[6]. The same procedure can be done more efficiently with divisions.

```
1: procedure EUCLID(m, k)
2:     if k == 0 then
3:         return m
4:     else
5:         return EUCLID(k, m  mod k)
```

The derivation of G. Touissant[1] comes from an analyisis over Bjorklund's studies on SNS accelerators[7] in nuclear physics. In this case, time is divided into intervals and during some of these intervals an onset is to be enabled by a timing system that generates pulses that accomplish this task. The problem for a given number n of time intervals, and another given number $k < n$ of pulses, is to distribute the pulses as evenly as possible among these intervals.
In our case the algorithm to be develped take as input a tuple (k, n), where n represents the length of the played sequence and k the number of onsets inside the sequence. The output is a sequence of length n, where the k onsets are equally spaced inside the sequence. For example, if we consider the tuple $(4, 16)$, the result has to be a sequence like:

[1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0]

As Touissant explains in his work[1], with this mechanism we can produce all the possible rhythmic sequences, linking them to a specific musical genre(Figure 1). For instance:

- $E(2, 3) = [1, 0, 1]$ is a common Afro-Cuban drum pattern. For example, it is the conga rhythm of the (6/8)-time Swing Tumbao[8].
- $E(2, 5) = [1, 0, 1, 0, 0]$, when it is started on the second onset ($[1, 0, 0, 1, 0]$) is the metric pattern of Dave Brubeck's *Take Five* as well as *Mars* from *The Planets* by Gustav Holst[9].
- $E(5, 16) = [1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0]$ is the Bossa-Nova rhythm necklace of Brazil, started on the second onset[10].

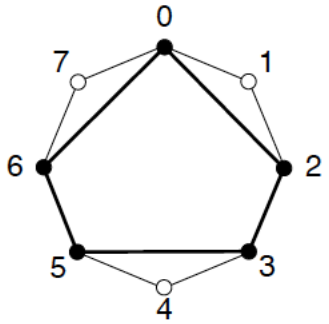

Figure 1: The Euclidean rhythm $E(5, 8)$ of the Cuban cinquillo

The application uses as reference the algorithm explained by Bjorklund[7]. It is composed by two steps: the first consists in creating the base string and the second in spacing the onsets inside the base string.
Considering m the length of sequence and n the number of onsets, and supposing that $n < \frac{m}{2}$, we need to create a sequence of $m - n$ zeros and one another of n ones. In the case of five pulses of thirteen cycles, we start from seqeuences [00000000] and [11111].
We start by simply dividing the five 1's into the eight 0's. This will put a 1 after every 0, with three 0's left over: [0101010101] and [000].
Once again, we can think of this as two optimally distributed strings. One strings is ten bits long and contains five "01" pairs (an optimal distribution of five pulses over ten slots). The second string contains three 0's. If we now distribute the three 0's over the five "01" pairs, we get three "010" triples with two "01" pairs left over: [010][010][010][01][01].
We still have two optimally distributed strings. "010010010" is an optimal distribution of three pulses over nine slots and "0101" is an optimal distribution of two pulses over four slots. We now repeat the process, dividing the two "01" strings into the three "010" strings. This gives us two five-bit strings ("01001") with one three-bit string ("010") as a remainder: [01001][01001][010][01].
We can stop the process when the remainder reaches one or zero. Our final pattern, therefore, is [0100101001010] , which is as evenly as we can distribute five pulses over thirteen slots.

3 The application

3.1 Sequencing

3.2 Synthdefs

3.3 Graphical User Interface

The Gui is divided in two main sections:

- The instrument grid
- The side bar

The Instrument Grid is composed by an arbitrary number of slots, which can contain an instance of an instrument. The type of instrument can be selected through a drop down menu.

Each instrument, shows some specific parameters that act on the temporal and timbral characteristics of the sound, and a sequencer section that is common among all the instruments.

The sequencer section contains all the parameters useful for setting an euclidean sequence to the selected instrument, a humanizer knob, and a volume knob. As far as the Euclidean sequence is concerned, the

step knobs, sets the length of the rhythmic sequence in terms of sixteenth notes; the hits knob sets the number of onsets, to distribute as evenly as possible in the sequence, according to the Euclidean algorithm. Finally, the offset knob allows the sequence to rotate, to achieve all the possible rotation of the same Euclidean rhythm. The humanizer knob can be used to specify the amount by which every onset in the sequence is shifted apart with respect to their precise location, in order to simulate a human behaviour. The volume knob simply allows to adjust the amplitude of each sound generated.

The Side Bar is used to set the global tempo in beat per minutes.

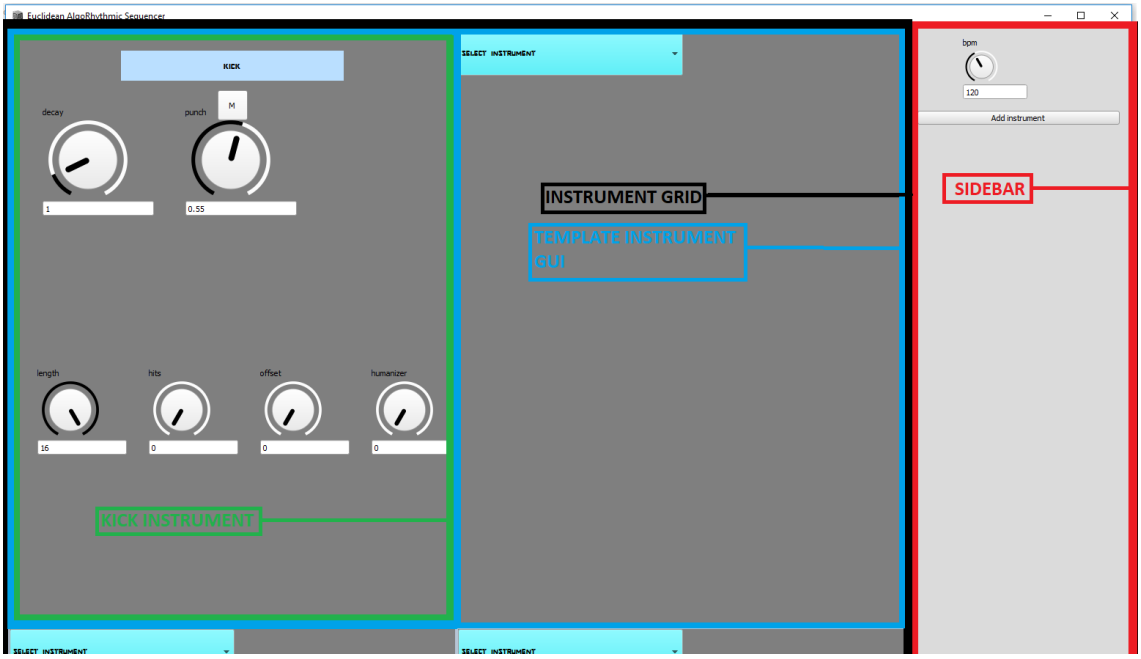

Figure 2: View of application

The graphical user interface has been developed following two guidelines: to be responsive and dynamic.

The responsiveness of a GUI means its adaptability to different screen sizes and formats. In order to achieve this goal, it has been decided to make a strong use of the composite pattern implemented in Supercollider. The composite pattern is characterized by a given class A, that can contain a list of other instances of the same type A, thus creating a three-like structure. As an example, the root could be the application window, that contains other groups of elements, that can contain themselves other elements and so on... Supercollider implements the composite pattern through the View class, in particular each GUI object must belong to a subclass of View.

View requires two arguments, the container of the element, and a set of parameters describing its bounds in term of pixels.
In the actual implementation each GUI component passes itself to the constructor of the children components, so that they can infer its dimensions and scale their size accordingly.
In the code snippet below, it is shown an example of this recurring pattern: a container, called instGrid, passes itself to its child component TemplateInstGUI. This behaviour is needed, because all the classes in the application that create GUI components, being subclasses of View, need the container View on which they must be attached to. It is also useful for the scaling purposes explained before.

Listing 1: Generate View from main.scd

```
1 (
2 s.waitForBoot{(
```

```
//Code from main.scd

159 instGrid = ScrollView.new(w,Rect(0,0, instGrid_width, w_height) );
//...

173 slot1 = TemplateInstGUI.new.create(instGrid);
```

Listing 2: Controller of TemplateInstGui

```
1  TemplateInstGUI{
2          var m, templateView;

4          create{ arg instGrid;

6                  var instGrid_width = instGrid.bounds.width;
7                  var templateView_spacing = 7;//spacing between the various
                       templateViews

9                  //CREATES THE CONTAINER "templateView"
10                 templateView = CompositeView.new(instGrid, Rect(0, 0, (
                       instGrid_width/2) - templateView_spacing, (instGrid_width*2/3) -
                        templateView_spacing ) );
                   //...

30         }
31 }
```

To ensure complete scalability of the GUI, in the main.scd script, the sizes of the screen are retrieved through a system call, and those quantities are used to scale every other portion of the GUI. Going up in the hierarchy of components and subcomponents we reach the main window that contains the whole application, which is proportional to the screen size. This implementation allows to have a user interface that scales with the screen dimensions.

In order to have a dynamic GUI, that allows the user to create, erase and replace instances of the various instruments, it was necessary to define a slot, which could host any different kind of instrument. Consequently, a way to abstract on the way each specific instrument is created, was needed.
The proposed solution makes use of the factory method patter, which is a common design pattern in object oriented programming. This enables writing of subclasses to change the way an object is created (to redefine which class to instantiate). To implement it, it is necessary to make an abstraction on the way instrument objects that are created (Inst class) and provide a class that is responsible for the creation of such objects (Inst Factory). The class diagram below depicts the simplified architecture for instruments creation and behaviour. The Supercollider classes are omitted for the sake of clarity.

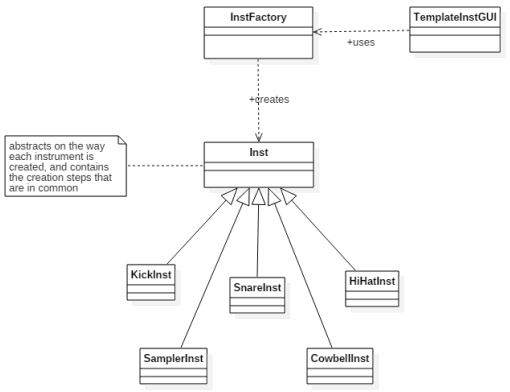

Figure 3: Class diagram of the application

To show the advantages of such pattern, it will be shown how TemplateInstGUI is able to create and "host" a generic instrument, by using InstFactory, without worrying about the differences in the creation of each specific instrument. Since Supercollider does not have interfaces, like most object oriented programming languages, the abstraction is done by dividing the common creational step among the various instruments in the method initializeSequencerGui in the superclass Inst. This method is then called during the creation of instrument specific GUI elements in the createView method, in each instrument class. Here only the code for the kick instrument is shown, since the other instruments present a very similar code structure.

Listing 3: TemplateInstGui Class

```
1  TemplateInstGUI{
           //...

5          create{ | instGrid |
                   //...

15                 m = PopUpMenu(templateView, Rect(0, 0, instGrid_width/4,
                       instGrid_width/20));
16                 m.items = InstFactory.getInstuments;
                   //...

22                 m.action = { arg menu;
23                         if(menu.item!="SELECT INSTRUMENT",
24                                 {var inst = InstFactory.getIstance(menu.item);
25                                         inst.createView(templateView);
26                         }, {});
27                 };
28         }
29 }
```

Listing 4: Controller of TemplateInstGui

```
33 InstFactory {

35         classvar <>instrumentId=500;

37         *getIstance{ | name |

39                 var instance = case

41                 { name == "kick" }    { instance = KickInst.new}

43                 { name == "snare" } { instance = SnareInst.new }

45                 { name == "hi-hat" } { instance = HiHatInst.new }

47                 { name == "sampler" }    { instance = SamplerInst.new }

49                 { name == "cowbell" }    { instance = CowbellInst.new };

51                 instance.pdefId = this.prGenIstrumetId;
52                 ^instance;

54         }
           //...

67 }
```

Listing 5: Instrument Abstract Class

```
69 Inst {
```

```
        //...

77      /*initializeSequencerGui INITIATES THE GUI RELATIVE TO A GENERIC INSTRUMENT
            */
78      initializeSequencerGui { | instView |
                //SETTING COMMON GUI DIMENSIONS
                //SETTING INSTRUMENT LABEL
                //CREATING EUCLIDEAN RHYTHM CONTROLS
                //CREATING HUMANIZER CONTROLS


166     }
        //...

194 }
```

Listing 6: Kick Class

```
197 KickInst : Inst {
        //...

202     /*CREATES THE INTRUMENT SPECIFIC GUI COMPONENTS, AFTER CREEATING THE
            COMPOSITE VIEW FOR THIS INSTRUMENT*/
203     createView{ | templateView |

205             instView = CompositeView.new(templateView, Rect(0, 0, templateView.
                    bounds.width, templateView.bounds.height) );
206             instView.background = Color.grey;

208             super.initializeSequencerGui(instView);
                //...


240     }
        //...

256 }
```

4 Conclusions

conclusions....

References

[1] Godfried Toussaint, "The euclidean algorithm generates traditional musical rhythms," Tech. Rep., School of Computer Science, McGill University, Montreal, Quebec, Canada, 2004.

[2] O. Gillet, "Mutable instruments. yarns, eurorack for generetion of euclidean rythms," Available at https://mutable-instruments.net/modules/yarns/manual/.

[3] Vladimir Pantelic, "Euclidean circles eurorack," Available at https://vpme.de/euclidean-circles/.

[4] "Modulus for generetion of euclidean rythms," Available at http://www.twohp.com/modules/euclid.

[5] J.-P. Allouche and J. O. Shallit, Eds., *Automatic Sequences*, Cambridge University Press, 2002.

[6] Euclid, *Elements*.

[7] E. Bjorklund, "The theory of rep-rate pattern generation in the sns timing system," Tech. Rep., Los Alamos National Laboratory, Los Alamos, U.S.A, 2003.

[8] Tom Klower, *The Joy of Drumming: Drums and Percussion Instruments from Around the World*, Binkey Kok Publications, Diever, Holland, 1997.

[9] Michael Keith, *From Polychords to Pólya: Adventures in Musical Combinatorics*, Vinculum Press, Princeton, 1991.

[10] Gerard Behague, "Bossa and bossas: recent changes in brazilian urban popular music," *Ethnomusicology*, vol. 17, no. 2, pp. 209–233, 1973.