The next two sections will present the graphical user interface in term of functionalities and architectural paradigm.

The GUI is divided into 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.

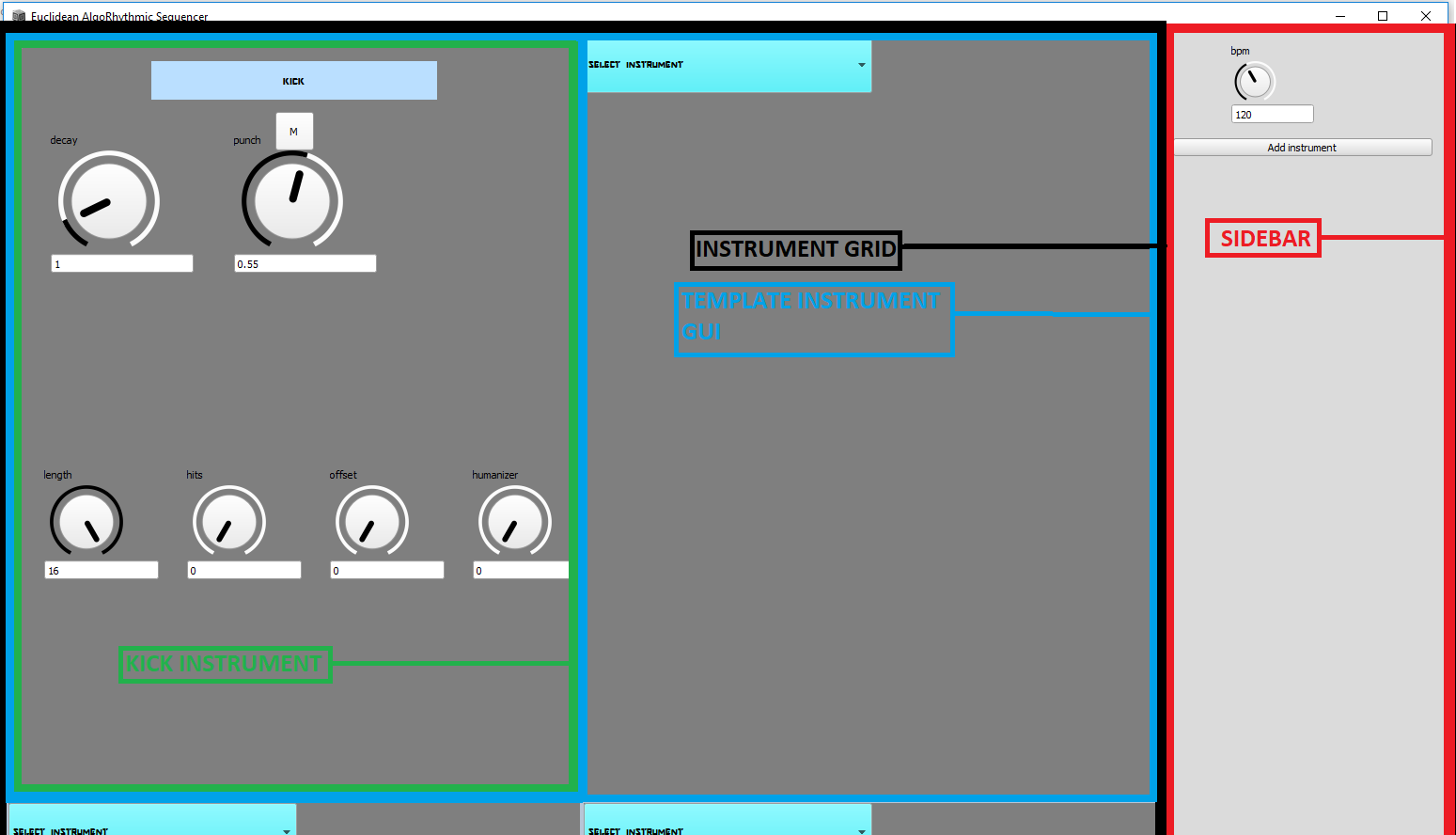
The sequencer section contains all the parameters useful for setting an euclidean sequence to the selected instrument, a humanizer knob, and a volume knob.

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.



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.

/code

//in main.scd file:

instGrid = ScrollView.new(w,Rect(0,0, instGrid\_width, w\_height) );

…

slot1 = TemplateInstGUI.new.create(instGrid);

//in instruments.sc file

TemplateInstGUI{

var m, templateView;

create{ | instGrid |

var instGrid\_width = instGrid.bounds.width;

var templateView\_spacing = 7;//spacing between the various templateViews

//CREATES THE CONTAINER "templateView"

templateView = CompositeView.new(instGrid, Rect(0, 0, (instGrid\_width/2) - templateView\_spacing, (instGrid\_width\*2/3) - templateView\_spacing ) );

…

}

}

…

/endcode

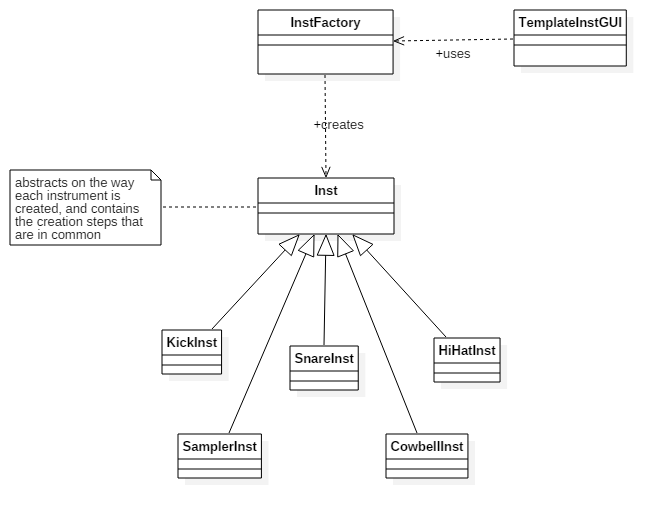
To ensure complete scalability of the GUI, in the main.scd script, the sizes of the screen are retrieved through a system call, and those quantities are used to scale every other portion of the GUI. Going up in the hierarchy of components and subcomponents we reach the main window that contains the whole 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.



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.

/code

TemplateInstGUI{

…

create{ | instGrid |

…

m = PopUpMenu(templateView, Rect(0, 0, instGrid\_width/4, instGrid\_width/20));

m.items = InstFactory.getInstuments;

…

m.action = { arg menu;

if(menu.item!="SELECT INSTRUMENT",

{var inst = InstFactory.getIstance(menu.item);

inst.createView(templateView);

}, {});

};

}

}

InstFactory {

\*getIstance{ | name |

var instance = case

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

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

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

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

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

instance.pdefId = this.prGenIstrumetId;

^instance;

}

…

}

Inst {

….

/\*initializeSequencerGui INITIATES THE GUI RELATIVE TO A GENERIC INSTRUMENT\*/

initializeSequencerGui { | instView |

//SETTING COMMON GUI DIMENSIONS

//SETTING INSTRUMENT LABEL

//CREATING EUCLIDEAN RHYTHM CONTROLS

//HUMANIZER CONTROL-----\*/

}

}

KickInst : Inst {

…

/\*CREATES THE INTRUMENT SPECIFIC GUI COMPONENTS, AFTER CREEATING THE COMPOSITE VIEW FOR THIS INSTRUMENT\*/

createView{ | templateView |

instView = CompositeView.new(templateView, Rect(0, 0, templateView.bounds.width, templateView.bounds.height) );

…

super.initializeSequencerGui(instView);

/\*---------------INSTRUMENT-SPECIFIC GUI ELEMENTS--------------------\*/

….

}

/end code

HUMANIZER: