Graphic rendering of audio processing specification v.1.08

Grame
Centre national de création musicale

April 21, 2015

Introduction

The aim of this specification is to define the possible representations of musical processes through INScore objects. For now, it is already possible for the score components to evolve following time messages, such as a clock or a piece of music, but this remains only possible for the synchronized objects, and they are simply moved (sometimes stretched) to fit their master's mapping. All other informations than time are not represented. We now would like to imagine an object changing size, orientation, color, transparency, etc. following the values of a signal. Therefore, we will first expose the existing tools to create signals in INScore and then see how we could use them to control any characteristic of any object following a signal.

1 The signals in INScore

As said before, it is already possible to represent incoming signals in INScore, but they are only meant to be drawn as graphical signals and not to control arbitrary components'properties.
For now, any object possesses a static node that can be seen as a container of all signals possibly present in the object. Those signals can be simple or more complex (parallel signals) and they are read by a graphical signal on the same level of hierarchy than our signal node.

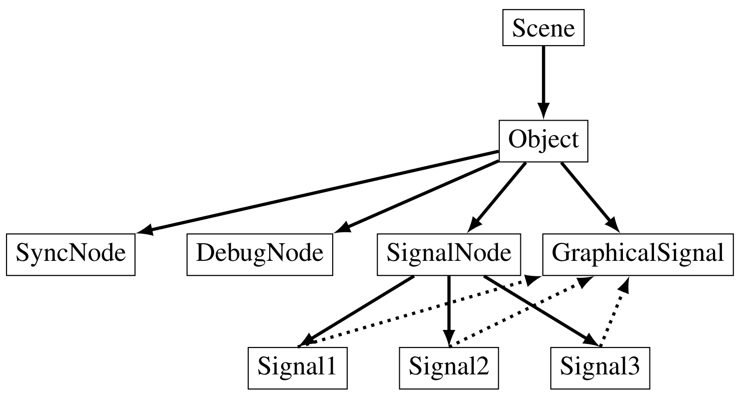

Figure 1: The Signal Node

2 Linking a signal to an object's property

There could be several ways to define the connection between signals and attributes in INScore. First of all because a signal can have a dimension greater than 1 (composed of several simple signals) and the attributes as well (like a color - H, S, B, A or R, G, B, A - or a position - x, y, z).

We could imagine the largest and most permissible syntax possible : that would be linking a $n-dimensional$ signal to a $m-dimensional$ attribute, and handle the different cases for $n \neq m$.

But, considering there are really few $m-dimensional$ attributes, and that each of those can be decomposed in "sub-attributes" (for instance R, G, B, and A for a color) that can me modified independently, we saw no reason why the notations should be more complex, when we could choose to simplify it at the most : One signal connected to one or more attribute(s) with a single dimension (Figure **??**). The color and the position will then have to be linked for each "sub-attribute", and not as a whole.

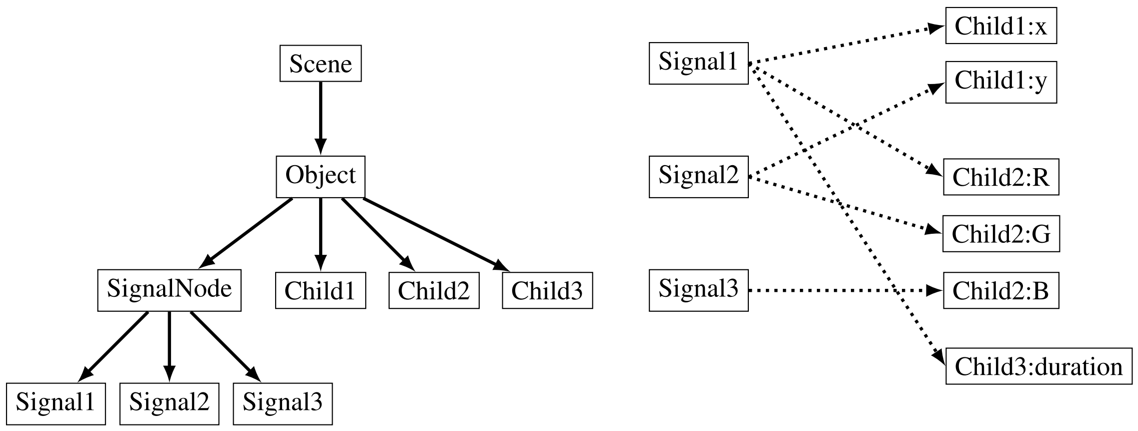

Figure 2: Example of connections between signals and attributes

The signal node will receive the connection messages and store all the links between signals and attributes in a map of the form :

```
std::map < std::string "object:attribute", ParallelSignal signal >
```

This form has been chosen in order to assure the unicity of the map key.

The syntax chosen for the OSC message is :

Where r1 and r2 are optional range bounds to specify the range of our values if different from the default ones. They will be stored in a map of the form :

```
std::map < std::string "object:atribute", std::string range >
```

Indeed, by default a signal will have a range of [-1, 1]. For some attributes, such as the position, the dimensions or the scale, those values stay exactly the same, because they correspond to our graphical space.

Nonetheless, there are some attributes that must have another default range, the [-1,1] range having no meaning for them : the date, the duration, the angle, the color...

That is why the bounds `r1` and `r2` will allow us to change the range to fit the incoming signals or the dimension of our attributes. For example, if a cursor is synchronized on a score, we could set the range of variation of our cursor's date to the score duration, so that the values of the signal can make it move on the whole score.

The connection mechanism can be sum up as follows :

- `MsgHandler::msgStatus ISignalNode::connectMsg (const IMessage * msg)`
  - Looks for the signal (first parameter of the message) in its list,
  - For each other parameter (of the form "`object:method1[]:method2[]`...") :
    - Separates the object from its methods,
    - Calls `connect ( sig, objectName, methods)`.
- `MsgHandler::msgStatus ISignalNode::connect (SParallelSignal sig, string object, string methodList)`
  - Dissociates all methods, and each method from its potential range,
  - Creates a new string "`object:method`" (for each method) and store it with the signal in the map `fConnections`, and also with its range in the map `fRanges`.
- `map< string, pair< SParallelSignal, string > > getConnectionsOf(string objectName)`
  - Looks for the object name in all connections,
  - Each time the name is found, we add the corresponding method, signal and range to the return map : `< method < signal,range> >`

3 Handling the connections

Like the OSC Messages that are computed by message handlers (`IMsgHandler`), we are going to create a signal handler class (`ISigHandler`), in order to call the corresponding setting method for each object's attribute.

The handlers and the methods available to connections will be stored in a new `fSignalHandlerMap` in the `IObject` class, similar to the `fMsgHandlerMap` or the `fGetMsgHandlerMap`.

Those handlers will be created at the object's construction and they will be called by its method :

```
IObject::executeSignal(string method, string range, ParallelSignal signal)
```

that will look for the right handler in the `fSigHandlerMap` and call its method :

```
virtual sigStatus operator ()(const ParallelSignal* sig, string range)
```

The attributes available to connections with signals are :

- width, height
- (d)scale
- (d)x, (d)y, (d)z, (d)xorigin, (d)yorigin
- (d)angle
- rotatex, rotatey
- (d)red, (d)green, (d)blue, (d)hue, (d)saturation, (d)brightness, (d)alpha
- (d)date, (d)duration
- pen(d)Alpha, penWidth

Some attributes will not be taken into account at first, but could be in the future, such as `clock`, or `show`.
The `clock` message does not take any value, and the `show` message expects a boolean.
In both cases, we could imagine a trigger that would send a `clock` message, or that would set the `show` message as `true` or `false`. The trigger value(s) could be set using the range parameters : when the signal values get out of these bounds, the message is triggered.