Augmented Score Library Specification v.0.52

Grame, Centre National de Création Musicale
<research@grame.fr>
ANR-08-CORD-010

September 16, 2013

1 Introduction

The Augmented Score Library [ASL] is intended to provide *augmented music score* capabilities. An augmented music score is a graphic space that supports music scores and arbitrary graphic resources, including real-time elements, and providing time synchronization between all the score components. For the time being, it is planned to support graphic resources such as textual elements, images (jpg, gif, tiff, png, bmp) and various signal representations. Time synchronization represents the possibility to graphically synchronize the augmented score components in order to align the graphic sections of the score components that correspond to equivalent time spaces. Time synchronization addresses an issue identified as *time to graphic mapping*, which is the core of the augmented score research project.

The ASL features are available using a standard C/C++ API or via *Open Sound Control*[**?**] [OSC][1] messages. There is a direct correspondence between the OSC messages and the library API. The global design of the library is based on the well known *Model View Controller* [MVC] approach.

2 MVC Design

The global architecture is illustrated in figure 1. Modification of the model state are achieved via the library API or incoming OSC messages. The modification requests are packaged into *messages* and stacked on a lock-free fifo messages stack. These operations are synchronous to the incoming OSC stream or to the library API calls.

On a second step, messages are popped from the stack by the `IControler`, which takes in charge address decoding and message passing to the corresponding object of the model. The model is organized as a tree, similarly to an OSC message and each node of the tree carries a name that should be used by valid OSC addresses. Address decoding consists in matching each part of the OSC address with the corresponding node until the last node of the address - the target node - is identified. The final step of the model modification consists in applying the message to the target object: the message is actually given to the `execute` method of the object. Each object of the model implements the necessary handlers to process its set of accepted messages. This second step of the model modification scheme is asynchronous: it is actually processed on a regular time base.

The final step of the processing scheme concerns the *view* update. It is implemented using the *visitor* design pattern [**?**]. An `Updater` provides the necessary to browse the model tree and call the derived `updateTo`

[1] http://opensoundcontrol.org/

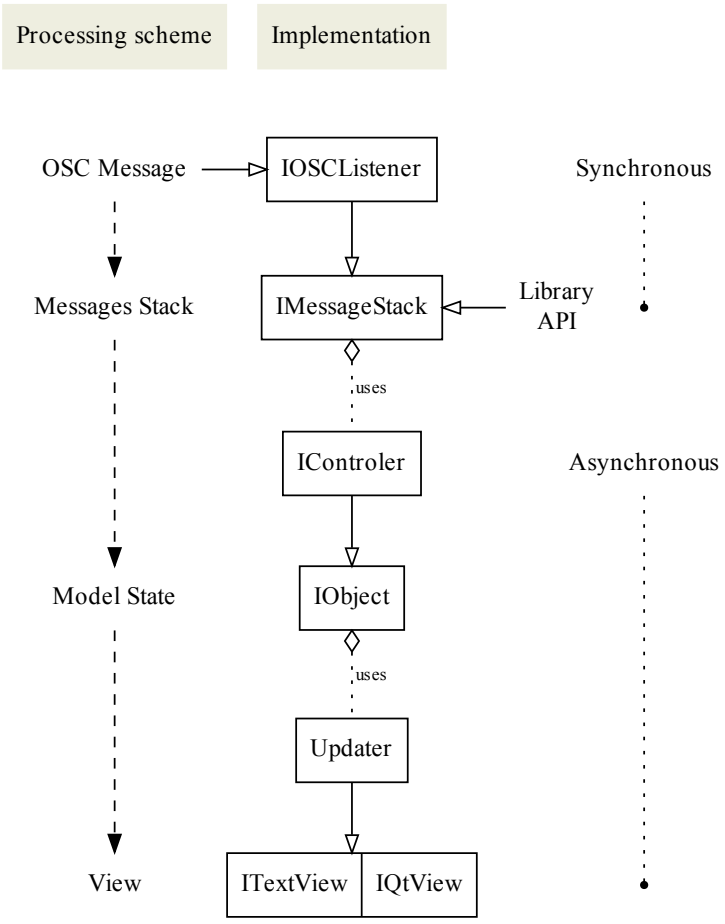

Figure 1: An overview of the MVC architecture

method only when necessary i.e. when the corresponding node is modified. An `Updater` is in charge of producing a view of the model. Two updaters are currently supported: an updater producing a textual view of the model (`ITextView`) and an updater producing a graphic view based on the Qt Framework (`IQtView`).

3 The Model Hierarchy

The figure 2 presents the model hierarchy. It is made of

- static objects: predefined objects that can't be created or deleted by client applications.
- dynamic objects: objects created by client applications.
- virtual objects: objects automatically embedded into another object.

Objects of the model are grouped by category:

- *Files* corresponds to file based components: an image file or a Guido GMN file.
- *Text* corresponds to textual components not based on a file.
- *Shapes* corresponds to vectorial graphics (ellipse, rectangle, polygon) and to a component named *Graphics* that takes a signal as input to produce a graphic representation of this signal.

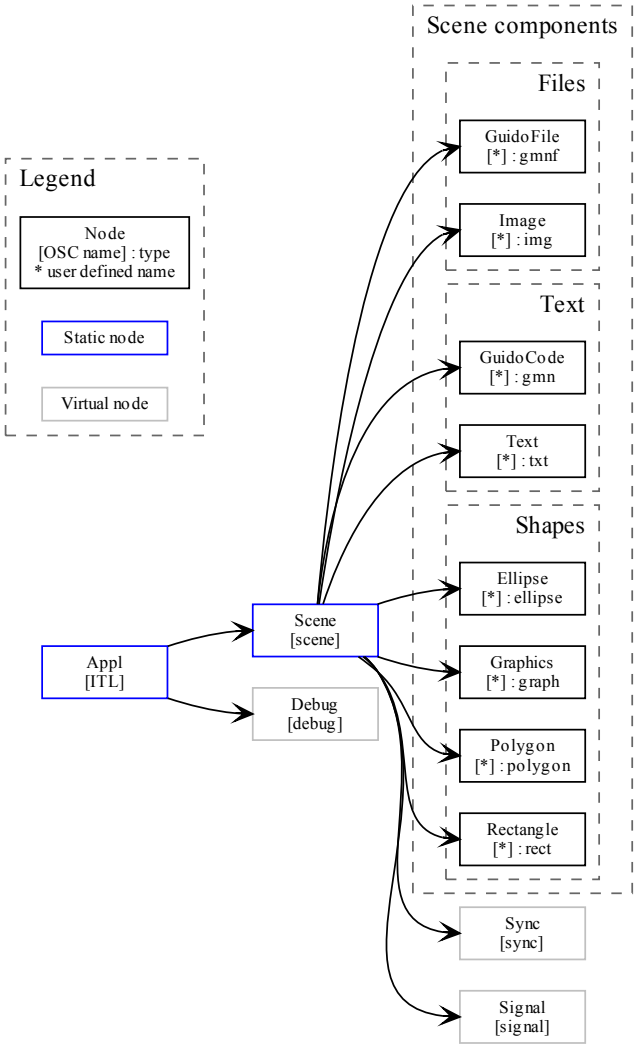

Figure 2: A non-exhaustive view of model hierarchy

Each object of the model has a name used as a identifier: static and virtual objects carry predefined names, all the other objects names are defined by the client application. A name must be unique for a given hierarchy level. The object's type mentioned in figure 2 is detailed in the OSC messages documentation.

4 The Update Process

The update process is actually composed of several steps (see figure 3):

- a first step consists in setting the model data according to the incoming messages
- the second step may be viewed as the partial integration of the view into the model: this step is intended to compute all the information necessary to process the next step, which is basically the graphic segmentation of an object and, depending on the object type, the necessary segmentations and mappings to go from the graphic to the time space [**?**].

- the third step computes the mapping that goes from an object local space segmentation to the corresponding graphic segments.
- the last step draw the objects according to the relations computed at the preceding step.

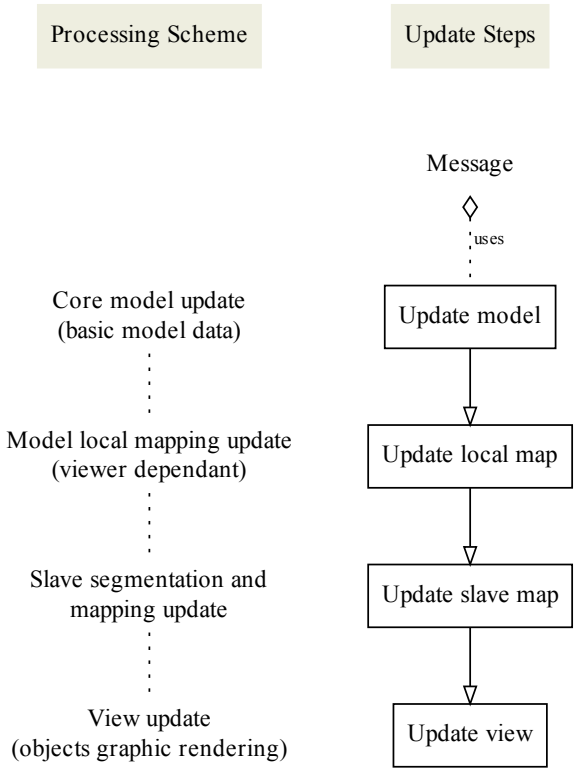

Figure 3: Details of the update process

All these steps are detailed in the next sections.

4.1 Core model update

to do... describe the messages processing scheme (handlers etc...)

4.2 Local mapping update of the model

Table 1 lists the segmentations and mappings used by the different component types. Mappings are indicated using arrows ($\leftrightarrow$). Note that the arrows link segments of different types (the *segment* qualifier is omitted). Segmentations and mappings in *italic* are automatically computed by the system, those in **bold** are to be provided externally.

The second step of the update process is in charge of computing the segmentations and mappings in *italic*. The computation is illustrated in figure 4, blue arrows denotes mappings to be computed by the local mapping update.

4.2.1 Events that trigger the local mapping update

A local mapping needs to be updated when:

type	segmentations and mappings required
text	*graphic* ↔ **text** ↔ **time**
score	*graphic* ↔ *wrapped time* ↔ *time*
image	*graphic* ↔ **pixel** ↔ **time**
vectorial graphic	*graphic* ↔ **vectorial** ↔ **time**
signal	*graphic* ↔ **frame** ↔ **time**

Table 1: Segmentations and mappings for each component type

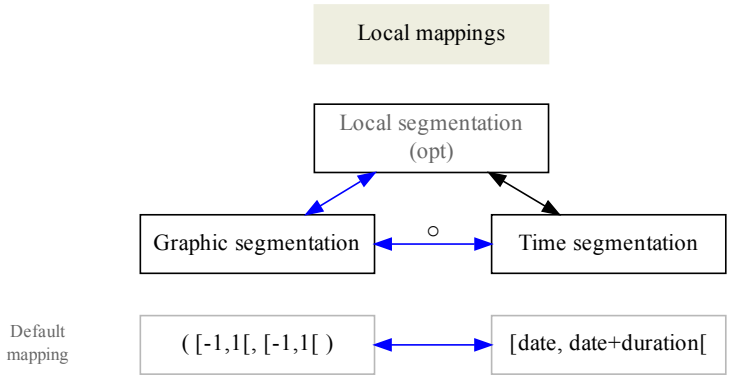

Figure 4: Local mappings computation: arrows in blue indicate the computations.

- the corresponding mapping provided externally has changed
- the object data has changed
- the object duration has changed and the duration is used for default mapping

4.3 Slave segmentation and mapping update

This step is intended to prepare the objects graphic rendering (update process, last step) and to make graphic synchronization transparent for the rendering process. The basic principle is to compute a mapping from an object local graphic space to its master graphic space.

Details of the mapping computation according to an object relations is given below. For the next sub-sections, we assume that the target object O of the update process has the following properties:

- it holds a time space segmentation $Seg(O_t)$
- it holds a graphic space segmentation $Seg(O_g)$
- it holds a mapping between graphic and time space segmentations $R_{O_{g<->t}}$

Next and when the target object has a slave relation with a master object M, we assume that M has the following properties :

- it holds a time segmentation $Seg(M_t)$
- it holds a graphic space segmentation $Seg(M_g)$
- it holds a mapping between time and graphic spaces segmentations $R_{M_{t<->g}}$

Note that the graphic space segmentation is expressed in the owner object local coordinates.

4.3.1 Mapping an object without synchronization

The update process has nothing to do. No mapping is used and the x, y and $scale$ properties are left unchanged.

4.3.2 Mapping a synchronized object without time stretching

The update process don't compute any mapping from local to graphic space but modifies the object x and y coordinates:

- it retrieves the segment M_{date} containing the object $date$ from $Seg(M_t)$
- it gets the corresponding graphic segment $M_g \in Seg(M_g)$ from $R_{M_{t<->g}}$
- and finally sets the object x to the corresponding position in M_g, obtained by linear interpolation and set y to the center of $M_g.y$ axis.

Note that the slave object coordinates are expressed in its master coordinates space.

Depending on the optional vertical stretch mode, the object $scale$ property is computed as the ratio between $M_g.y$ interval size and the object $height$.

4.3.3 Mapping a synchronized object with time stretching

The update process computes the mapping $R_{O_g<->M_g} = R_{O_{g<->t}} \circ R_{M_{t<->g}}$
Note that it may involve the computation of *virtual* segments since $Seg(O_t)$ and $Seg(M_t)$ don't necessary match. The virtual segments computation makes use of linear interpolation.

The computation is illustrated in figure 5, blue arrows denotes mappings to be computed by the slave mapping update. Note that it involves a virtual mapping since the slave and master segmentations don't necessary match.

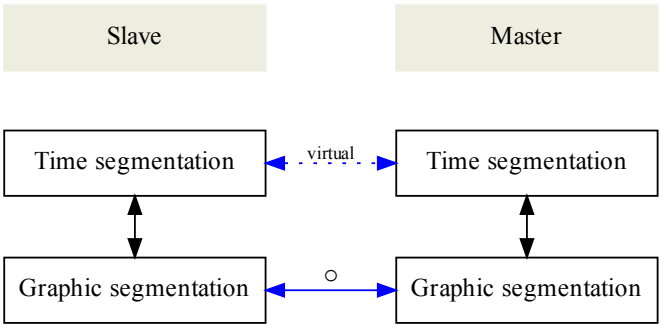

Figure 5: The slave mapping computation: arrows in blue indicate the computations.

4.3.4 Mappings translation

The $date$ of an object is used to translate the time segmentation of a slave object. The translation consists in shifting every time segment from the $date$ value.

4.3.5 Events that trigger the slave mapping update

The slave to master mapping needs to be updated when:

- the slave date has changed
- the slave or the master local mapping has been updated
- the synchronization mode has changed

4.4 View update

The view update consists in actual graphic rendering of an object. We make a distinction between the rendering of objects holding a mapping from local space to graphic space and those that don't:

- Rendering objects without mapping: the object is actually drawn using its current *x*, *y* and *scale* properties.
- Rendering objects with mapping: for each associated segments O_l and M_g taken from $R_{O_{l<->g}}$:
  - depending on the stretch mode, computes a *scale* as the ratio between $M_g.y$ interval size and the object *height*.
  - draw the segment O_l at the *x* and *y* coordinates taken from M_g and using the object *scale*.

References

[1] Wright Matthew. *Open Sound Control 1.0 Specification*, 2002.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[3] D. Fober, C. Daudin, Y. Orlarey, and S. Letz. Time synchronization in graphic domain - a new paradigm for augmented music scores. In *Proceedings of the International Computer Music Conference (submitted to)*, 2010.