----Title

OMlib - a simple and small event-driven framework for interactive music and installation

---

----What is OMlib?

OMlib is the collection of small C++ classes written to provide the beginner-class programmers with a simple and small framework that makes it much easier to build the interactive music/installation systems, currently under developping.

Since OMlib's basic architecture is based on an event-driven architecture, which is already familiar even to non-programmers as the one used in Java GUI and already has much popularity.

At the same time, the event-driven architecture is the one of the most appropriate architecture for an interactive music/installation, because of its nature of modeling that the architecture waits for a new event or a message which occurs and then interact.

Fully written as C++ class library, OMlib can provide the users with some remarkable features that the software like MAX or SuperCollider can not provide, like high portability, high reusability, high extensibility and so on

---

---- The problems in building the systems for interactive music and installation

---

The reason why OMlib was started developing is that some kind of the algorithms and new objects are hard to implemented on the major software like MAX or SuperCollider. Even though the applications like these provide users with rapid-application-development environment and software-development-kit so that users can build their own systems and if necessary, users can program and add their own new objects, the users who are willing to do so must have the good understanding not only of the programming language but also of the architecture of these applications. But it is an usual case that can be seen in the kind of the situations that the basic algorithms of the newly created objects can be easily programmed in the usual programming language like C or C++, if it does not require the interactive or real-time circumstances.

The main problem in creating the interactive music or installation is in the fact that the most of the musicians and artists are not well-trained programmers to create their own systems which can be hard to create on the existing ready-made software to accomplish what they want to do in their works by their own.

OMlib is supposed to be used such users who might be the musicians or the artists who have only very basic knowledge and limited experience in C++ and who are willing to build the systems which are little complicated to programmed on the existing ready-made software but not so if written in C++.

The main reason that makes it little harder to build the interactive systems for the beginner programmers than the other kind of the systems is that it requires the real-time handling of the events and the periodic time scheduling.

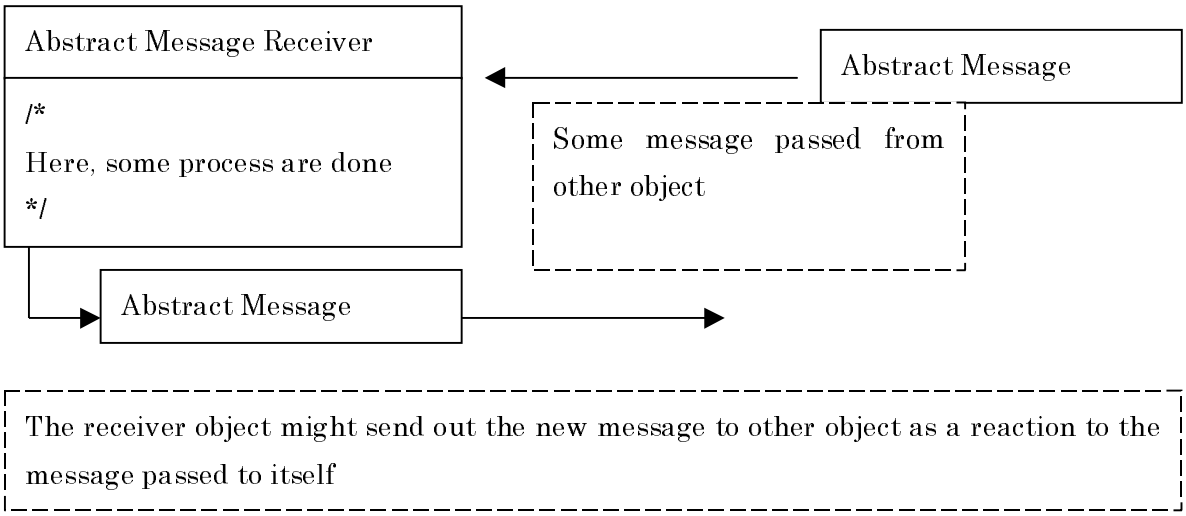
Thanks for the object-oriented nature of C++ and the event-driven architecture which is appropriate for an interactive system, OMlib did pack and hide the codes for real-time handling of the events and the periodic time scheduling and so on, which are little complicated and hard to be understand by the beginners

Since OMlib is a collection of small and simple C++ class library, not the software, it can provide the features that C++ provides like cross platform portability, high reusability. At the same time, not only being the class library but also a framework of the interactive systems for music and installation, OMlib makes the users share their accomplishments each other without difficulty. For instance, if a user creates the system that includes a simple motion capture based on USB camera, since OMlib provides a basic framework and unified interfaces between objects, it can be easily used by another user for his/her own new systems, even if the new user has really limited knowledge in writing codes.

----The basic architecture of OMlib

----1. two basic abstract classs

OMlib is based on the architecture of event-driven. There are only two main abstract classes the programmers mainly have to understand, basically. One is ' abstract message' class, and the other is 'abstract message receiver' class. Speaking generally, 'message' is data and 'message receiver' is something to process the data. If some programmers desire to build his/her own interactive system, he/she only has to understand this relation ship between these two classes. The figure1 below shows this relationship graphically.



(Figure 1)

All the message like note-on message that comes in from MIDI instruments or some byte-data passed from the serial port that connected to sensors are handled as 'message class' which is inherited from 'abstract message' class in OMlib. And, all the message classes are handled by the classes inherited from 'abstract message receiver.'

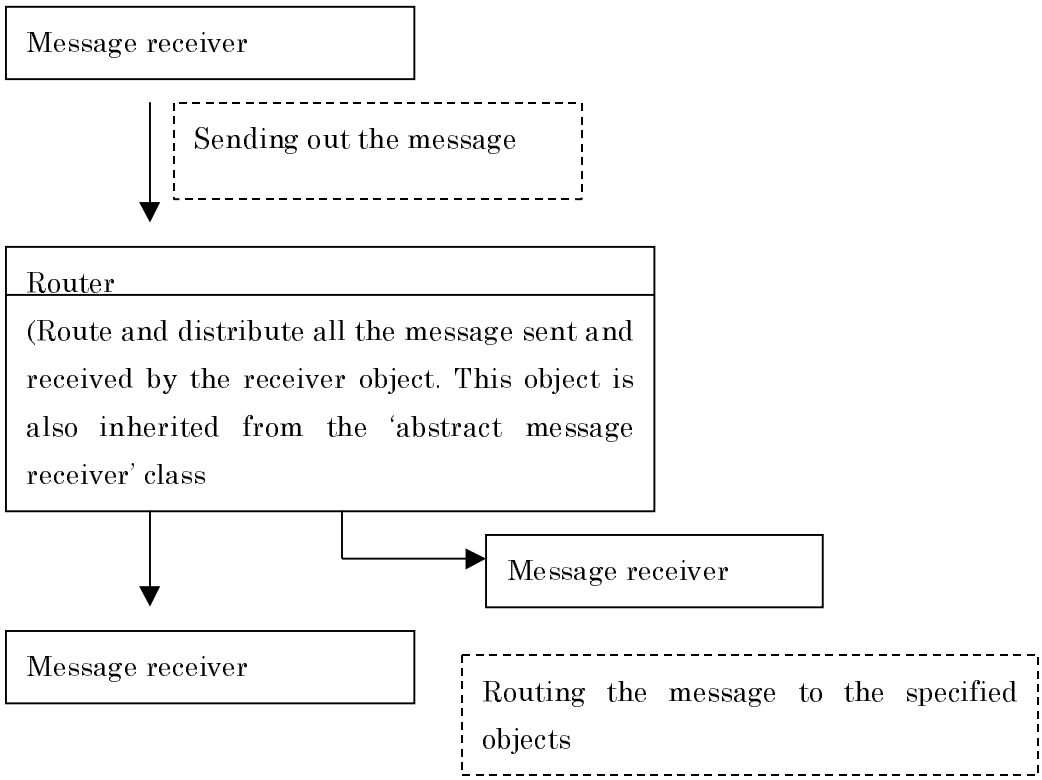
These mean that OMlib is providing with the unified interface and that as a result users can easily create his/her own object, inheriting these two abstract classes and add it to the systems they built. For example, if some user created a new object of temperature sensor, and replace it to the place where the MIDI input was there, the user only have to write a new object inherited from MIDI input class , which is also inherited from abstract Message receiver and just replace the new object and MIDI input class.

Speaking simply, when some event like MIDI input , the MIDI input object, which is child class of the abstract message receiverclass, send out the message like note-on,

which is the child class of the abstract message class. (Event occurs and the message is sent out.) then the other message receiver class, for instance suppose a MIDI-pitch shifter object, receives the message.(From the view point of the pitch-shifter class, the event was triggered by the note-on message). This event driven architecture is appropriate for interactive works because the most of interactive works need to react the event dynamically in real-time, the situation of which is difficult to handle in the other type of modeling.

----.2 .routing architecture

OM lib also provides with 'routing ' architecture like the one used in VRML. The 'routing' architecture is used to let message receiver objects exchange the messages, being hidden form others, without knowing the objects to receive the message and provide the one and only passage where the message go through. (See figure 2 below)



(figure 2)

Thus, 'routing' architecture hides the objects each other and at the same time make it

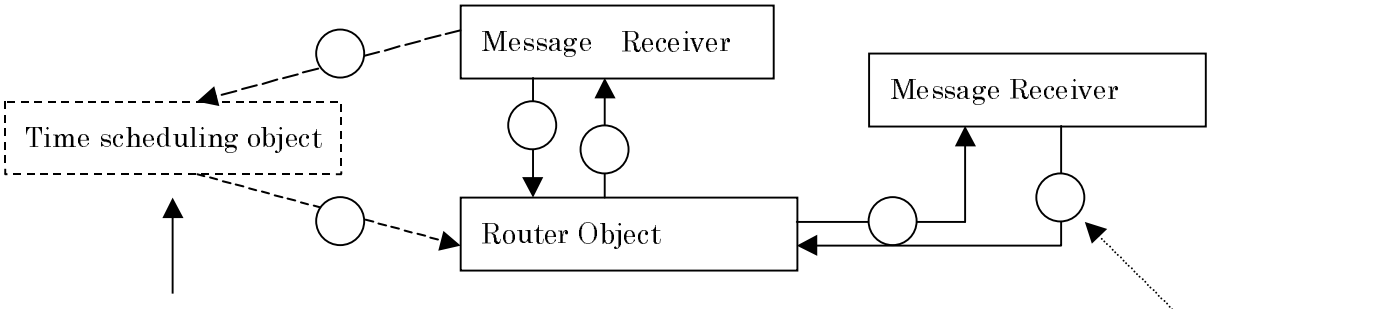
able to distributing the message to multiple objects. This routing architecture has an advantage that the possibility of reusability is increased much more.

Since an object is hidden from others and the router object is one and only passage that the message go through, the change of one receiver object never effect the other objects and the programmers does not have to rewrite the code of other objects and just have to rewrite to replace the object with the new object. The lines that are needed to rewrite might be much less than when the routing architecture is not provided.

At the same time, since routing architecture limits the path of the message, the programmers can write the codes in a much effective and simple way. This directly means less bugs in the code.

----3 the periodic time scheduling

The one of the characteristic problem in the interactive music/installation is the periodic time scheduling. Without periodic time scheduling, the interactive system hardly can handle the precise delay or metronome-like function. OMlib encapsule the time scheduling function and the users doesn't have to know the detail and easily can access such function like delay and metronome.



The scheduling object is completely hidden from the other objects so that the programmers do not have to think of the actual time scheduling process and the messages can be sent as if there are no such thiing as the time scheduling object.

The messages sent to/from the other objects

----Progress situation

OMlib is currently under developing. But the basic parts are already finished and now the additional objects like TCP/IP connection and simple video sensor USB camera are being developed . The additional sensoring devices might be added later.