Coupling Virtual Reality Open Source Software Using Message Oriented Middleware

Fábio Rodrigues, Rodrigo Ferraz, Márcio Cabral, Fernando Teubl, Olavo Belloc, Marcia Kondo, Marcelo Zuffo, Roseli Lopes
Laboratory of Integrated Systems, Polytechnic School, University of São Paulo
São Paulo, SP - Brazil
{fabioldr, rferraz, mcabral, ftf, belloc, mkondo, mkzuffo, roseli}@lsi.usp.br

Abstract

There is a multitude of open source software API's for developing Virtual Reality applications. These API's are useful and great tools for newcomers as they provide a fast learning curve and cover a broad range tasks. However, most of the time, each one of these API's is focused on a specific task, such as scene graph, physics simulation and input device handling. Building an application using these different API's together, results in a solution that lacks software engineering and compromises future maintenance and reuse. This paper presents the concept of Message Oriented Middleware (MOM) to encapsulate Virtual Reality Software API's. Our approach provides an organizational in tiers that allows message exchanges in a real-time Virtual Reality application built upon multiple API's. We present results from a validation implementation for a multi-screen cluster powered display.

KEYWORDS: Message Oriented Middleware, Software Encapsulation, Virtual Reality.

INDEX TERMS: 5.1.b [Information Interfaces and Representation]: Artificial, augmented, and virtual realities; D.2.1.e [Software Engineering]: Methodologies; D.2.2.d [Software Engineering]: Modules and interfaces

1. Introduction

The Virtual Reality (VR) community commonly uses open source software to create virtual reality applications. Many of these solutions are based on coupling different software or library API's to achieve the desired result.

Based on our experience, we observe that such development method reduces implementation time and rapidly provides an initial instance of the final target application. However, the resulting application posses several limitations, described below:

- Lack of a common well defined software architecture: each software API has its own requirements and dependencies, imposing restrictions on software architecture and organization;
- Little or no code reuse: coupling is usually restricted by application features;
- No extensibility: tight coupling design restricts future changes in order to achieve the intended target application.

Developers can focus on application logic and functionalities using a message middleware layer as coupling subsystem, assuming that the task of software encapsulation will occur just once and then reused.

Communication between encapsulated software can also be easier and more flexible in a message exchange scenario, where multiple types of messages coexist, allowing developers to choose which types of message to use based on implementation requirements.

If the message middleware can use flexible communication channels with minimum required setup, the deployment and maintenance of many VR applications will be facilitated, especially when we consider a distributed cluster environment.

The ability to couple different software in the same process is also a desirable feature considering small and even embedded applications.

In this paper, we propose an application architecture using Message Oriented Middleware (MOM) as a coupling subsystem to aggregate different software entities in order to create complex Virtual Reality application. We also developed a demo application to preliminarily validate the architecture and our middleware implementation.

2. Related Work

Several frameworks for virtual reality software have already been developed [1][2][3][4][5]. Some of these systems [3][4] are based on data flow paradigm, which divides the whole application in several modules connected by its inputs and outputs in a complex network. On the other hand, other systems [1][2] implement event-based architecture where the application is also organized in modules, but the data transfer between modules is triggered by events.

In this paper, we propose a framework for open source software integration based on loose coupling architecture using extensible message oriented middleware.

InTml [3] is a XML based language that describes a VR application in terms of input/output devices, interaction techniques and geometries.

FlowVR [4][5] is a data flow oriented middleware for distributed VR applications composed by modules as separated SO processes running on local or on remote computers.

VR-DECK (Virtual Reality Distributed Environment and Construction Kit) [1] is a toolkit for building distributed virtual worlds.

Each module is defined by rules that describe input events, output events and actions. Modules are independent processes, as in FlowVR, exploring the maximum of parallelism. However, light modules cannot be grouped under the same process, causing a lot of context switches.

The HIVE [2] is another event-based framework for the development of virtual environments. It is based on publisher/subscriber architecture. The communication between modules is over UDP and multicast network.

Differently from the implementations cited above, our framework focuses on providing easy-to-setup and medium-independent communication.

3. Application Architecture

In this architecture, each software entity is treated as a module with its specific functionalities inherited from the encapsulated software. More than one module can coexist in the same process. Modules communicate with each other by sending messages asynchronously through the message middleware.

Modules define the types of messages they send and broadcast registration messages with messages types they receive. Based on correlation between senders and receivers logical connections are established.

Modules don't share much information from other modules; they ought to only coexist in the same communication medium and be aware of message interfaces, creating a loose coupling relationship.

Messages types follow modules functionalities; similar purpose modules should send and receive an analogous set of messages, thus creating similar interface and easing cross-compatibility.

Figure 1 shows a hypothetical case of development through bounding different software modules would achieve an application with interaction, visualization and audio. Although the framework is not limited by this set of modules, the example shows the interaction between modules on the complete application architecture.

Each module can be any software that implements the functionality described.

In the subsections below, each entity showed in figure 1 will be explained in detail.

3.1. Scene Graph Module

This module could be any scene graph such as Ogre [13], Open Scenegraph [8], OpenSG [7] etc. This freedom of choice enables to select the software with features that best addresses the application requirements.

The types of messages that the scene graph interprets are related to visualization, *e.g.*, camera positioning, object selection and object movement messages. If the application does not require any special functionality from the scene graph, the library can be seamlessly swapped without reprogramming any other module.

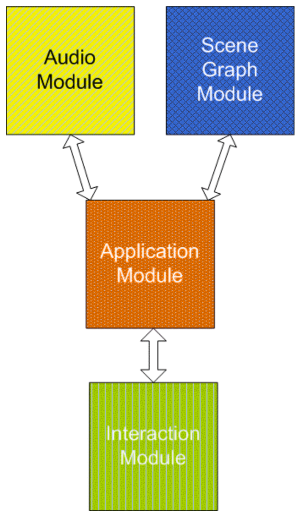

Figure 1. Example application diagram

3.2. Interaction Module

The Interaction Interface is the module responsible for encapsulating the functionalities of interaction devices such as joysticks, data gloves and trackers.

This module can be implemented by an API that encapsulates functionalities from libraries such as Glut [11] or SDL [15]. Even a more complex interaction system, such as ARToolKit [14], can be encapsulated.

Interaction related messages such as 3D pointer position, avatar movement, selection commands etc. should be cross-compatible between different encapsulated software to act as the interaction module.

3.3. Audio Module

Any audio library, such as OpenAL [9] or ALSA [10], can be encapsulated to create an audio module.

The audio module should interpret messages regarding movement, sound position and triggering audio clips.

3.4. Application Module

This module implements the application logic. It is responsible for all features required by the application.

In this development scenario where all software entities have been previously encapsulated, the application module is the only software that would be implemented. This would help the development team to focus only in the application logic and its functionalities, without worry about API incompatibilities and integration issues.

4. Coupling Middleware – Message Manager

The middleware was created to be cross-platform software that enables medium free communication between software modules.

In order to enable loose coupling between software modules, we developed an event based MOM called Message Manager. The Message Manager provides communication between software modules running within the same or in remote processes.

Message Manager allows communication between modules in the following cases:

- Modules within a process
- Modules in different processes running in the same computer
- Modules running in different computers even with different operational systems
- A hybrid combination of cases described above.

4.1. Message Manager Operation

A Message is an entity composed by a type (Message Type) and a list of parameters. When modules send messages, they define the type and add parameters to them (they compose the message). Each parameter has a name and a type. Currently the supported types are the C++ native types (int, float, char, etc.) and strings and arrays of these types.

Message Manager uses the message type (a C++ string) to route and deliver the message to its correct receivers. Message types are hierarchically organized in tree format. For example, when the mouse is moved towards the *y*-axis, the message generated could have the type *input.mouse.move.y*, with the *y* coordinate as the parameter. For the *button 1* pressed, the type could be *input.mouse.button.1*. This way, the receiver could be registered to specific types (*e.g. input.mouse.move.y*) or group of events (i.e. *input.mouse*, for all mouse events).

Receivers are objects that can be registered to receive messages. To be registered, the class should inherit the *Receiver* abstract class and implement the *OnMessage* virtual method. Message Manager uses this method to deliver messages to objects within a module calling the *OnMessage* method and passing the message reference as a parameter.

Receivers are registered for specific message types in runtime. In contrast to publisher/subscriber paradigm, in our approach, only receivers (subscribers) should be registered. All entities in a module can compose and send messages through the Message Manager API.

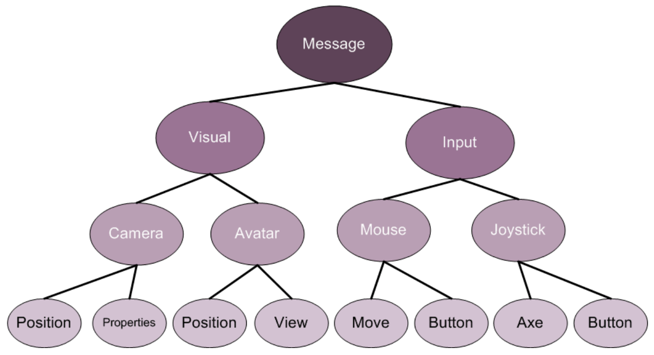

Figure 2. Registry Hierarchical Organization

Regarding the application viewpoint, all receivers are connected by a logical network, called *Message Network*. When a sender puts a message on this network, all interested receivers will receive it.

In a lower level, a communication channel connects Message Managers running in different OS processes with flexible topology, such as: star, extended star, hierarchical, line or any hybrids of these topologies. Currently, the only restrictions are topologies with loops (mesh, ring etc.) because it could result in infinite message traffic.

Internally, when a message is sent, the Message Manager searches the registry for interested receivers. If a receiver is local (*e.g.* running on then same process) the *OnMessage* method is called. If the receiver is located on a remote process, then the Message Manager packs the message and sends to the remote Message Manager through a communication channel that handles messages. Therefore the message will only be packed and sent through the communication channel if a receiver exists on a remote Message Manager.

Currently the only supported communication medium is TCP/IP, but Message Manager can be extended to others like UDP, Myrinet, shared memory (for communication between process on the same computer) etc. The application is not limited to use just one type of communication medium at once; multiple medium can be used simultaneously. The *Factory* [6] pattern is used to instantiate the correct communication handler during runtime.

Message Manager also supports many packages formats. Currently only XML and basic binary formats are implemented. Other formats like compressed binary, encrypted, basic ASCII etc. can also be implemented. When a package arrives, Message Manager chooses the specific *unpacker* for it, based on the package header.

The Message Manager API is simple and easy to use. Four classes are available to the module:

- *Receiver*: It is the abstract class for the receiver's classes. All receivers should inherit this class and implement the *OnMessage* method
- *Message*: The message class. To create a new message, this class should be instantiated, the type set and the parameters added
- *MessageManager*: This is the facade [6] class to all operations over the *Message Network*
- *Address*: This class is a container for communication channel parameters. For example, if the communication channel protocol is TCP/IP, then the parameters will be the hostname and the port used for that connection

The *MessageManager* class has three kinds of operation:

- Management of communication channels (create and destroy)
- Send Messages
- Register/Unregister receivers

4.2. Message Manager Architecture

Figure 3 shows a simplified class diagram of the Message Manager. The four API classes (MessageManager, Receiver, Message and Address) are on top of the diagram.

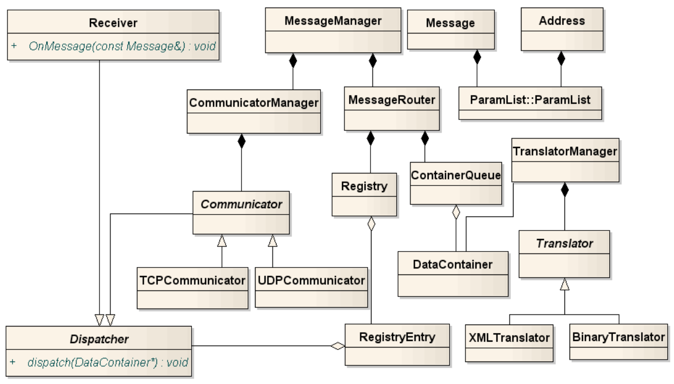

Figure 3. Message Manager Simplified Diagram

Message Manager operation is divided in three subsystems: Message Routing, Communication and Translation.

A message can be represented in several ways. In the application side, a message is represented by the high level *Message* class. But, to transfer a message through a communication channel, the data must be formatted according to the protocol used. To allow several representations, internally a message is represented by the *DataContainer* class. This class allows setting and getting the data as *Message* class or a *Package* class.

The *Message Routing* subsystem routes all incoming messages to local *Receivers* or to remote Message Managers, through communication channel. This subsystem has a message queue and a dispatcher registry. The last one associates dispatchers (internal interface for classes that receives *DataContainers*) with the type of messages that they receive.

The Communication subsystem manages the communication channels between Message Managers' remote instances. The *Communication* abstract class is used to handle the low level communication over a specific communication channel. Each supported communication channel has an implementation of this class.

The Translation subsystem translates messages to packages (in different formats), and vice-versa. Translators (classes that implements the *Translator* interface) create or parse packages within specific format.

The simplified flow when the application sends a message is:

1 – The message is converted to a *DataContainer;*

2 – The *DataContainer* is sent to the message queue (in Message Routing subsystem);

3 – During the Update cycle, the Message Routing subsystem checks the registry for dispatchers interested on this type of message, and calls the *dispatch* method;

4 – If the dispatcher is a *Receiver*, then *DataContainer* is converted to a *Message*. Else if, the dispatcher is a *Communicator*, then the *DataContainer* is converted to the correct Package and send to the remote Message Manager.

When a package is sent over a communication channel, then the flow is:

1 – During Update cycle, the *Communicator* receives the Package and converts it into a *DataContainer*

2 – The *DataContainer* is sent to the Message Routing subsystem message queue;

3 – The Message Routing subsystem dispatches the *DataContainer*, as described above.

5. Message Manager Validation - Demo Application

As a preliminary validation, we developed a demo application. This application is a 2x2 multi-projection powerwall, running a real time 3D model renderer application in a 5 node cluster. The nodes are connected using gigabyte Ethernet network.

An Intel dual core CPU with nVidia Quadro FX 4400 graphics card drives each projector in the multi-projection system. Each node runs a scene graph module based on OpenSG library, and renders only a part of the total image. A dedicated node runs the master module and a GLUT based input module, used to interact with the 3D model. The interaction module can drive other scene graph modules in then same *Message Network*.

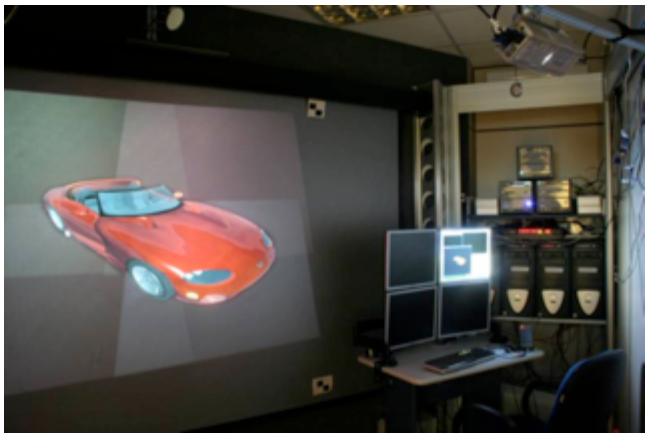

Figure 4 – Demo Application: Multi-projection Powerwall running real-time 3D model renderer

6. Conclusion and Future Work

Commonly, the development of VR application consists on coupling specific-propose open source software (*e.g.* scene graphs, input libraries, audio libraries) and implementing the application logic. If the core architecture is not well planed, then the software extensibility and reusability is compromised.

In this work, we presented a cross platform message oriented middleware called Message Manager as coupling subsystem to aggregate different software components and create complex VR applications. In this approach, third-party software entities are encapsulated in modules that can run in the same OS process or in remote process over message communication channels. Message Manager creates a simple logical message network, abstracting complex network topologies.

Different modules with the same propose send and receive the same message formats (type and parameters) in order to achieve loose coupling, easy reuse and application extension.

The demo application validated the concept of the architecture and the Message Manager implementation. However, more scalability and stress tests should be done.

To make Message Manager more robust and efficient, we will perform more scalability and performance experiments to optimize the message delivery algorithms, solve the looped topology issue, make it thread safe and implement new features, such as message priority and synchronous reply. In order to optimize the communication between Message Manager instances, other communication channels and package formats will also be implemented.

References

[1] Codella, C.F.; Jalili, R.; Koved, L.; Lewis, J.B., “A toolkit for developing multi-user, distributed virtual environments” *Virtual Reality Annual International Symposium, 1993., 1993 IEEE* , vol., no., pp.401-407, 18-22 Sep 1993.

[2] Zonghui Wang; Xiaohong Jiang; Jiaoying Shi, “HIVE: a highly scalable framework for DVE” *Virtual Reality, 2004. Proceedings. IEEE*, vol., no., pp. 261-262, 27-31 March 2004.

[3] Figueroa, P., Green, M., and Hoover, H. J. 2002. “InTml: a description language for VR applications”. In *Proceedings of the Seventh international Conference on 3D Web Technology* (Tempe, Arizona, USA, February 24 - 28, 2002). Web3D '02. ACM, New York, NY, 53-58.

[4] Arcila, T.; Allard, J.; Ménier, C.; Boyer, E.; Raffin, B., “FlowVR: A Framework For Distributed Virtual Reality Applications”, Premières Journées de l'AFRV, November 2006, Rocquencourt, France.

[5] Allard, J., Gouranton, V., Lecointre, L., Limet, S., Melin, E., Raffin, B., and Robert, S. 2004. “FlowVR: a Middleware for Large Scale Virtual Reality Applications”. In *proc. of the international conference EUROPAR 2004*, Springer Verlag, vol. 3149 of *LNCS*, 497--505.

[6] Gamma Erich, Helm Richard, Johson Ralph, Vlissides John: “Design Patterns, Elements of Reusable Object Oriented Software”, Addision-Wesley, published October 1994.

[7] OpenSG. http://www.opensg.org/.

[8] OpenSceneGraph. http://www.openscenegraph.org/.

[9] OpenAL. http://www.creativelabs.com/openal/.

[10] Alsa (Advanced Linux Sound Architecture). http://www.alsa-project.org/.

[11] GLUT. http://www.opengl.org/resources/libraries/glut/.

[12] FlowVR. http://flowvr.sf.net/.

[13] Ogre, http://ogre3d.org/

[14] ARToolKit, http://www.hitl.washington.edu/artoolkit/

[15] SDL, http://www.libsdl.org/