High Performance Interactive Computing with FlowVR

Jean-Denis Lesage *
INRIA
Laboratoire d'Informatique
de Grenoble (LIG)

Bruno Raffin †
INRIA
Laboratoire d'Informatique
de Grenoble (LIG)

Abstract

This paper introduces the FlowVR suite, a set of softwares targeted at high perfomrance interactive computing, in particular virtual reality applications, to be executed on clusters. The FlowVR middleware supports coupling of heterogeneous parallel codes and is component oriented to favor code reuse. After introducing the FlowVR main concepts, we details the different tools associated and several applications.

1 Introduction

Developing VR applications that include numerous simulations, animations and advanced user interactions is a challenging problem. We can distinguish two strong difficulties:

- Software engineering issues where multiple pieces of codes (simulation codes, graphics codes, device drivers, etc.), developed by different persons, during different periods of time, have to be integrated in the same framework to properly work together.
- Hardware limitations bypassed by multiplying the units available (CPUs, GPUs, cameras, video projectors, etc.), but with the major drawback of introducing extra difficulties, like task parallelization or multi devices calibration (cameras or projectors).

Software engineering issues have been addressed in different ways. Scene graphs offer a specific answer to graphics application requirements. They propose a hierarchical data structure where the parameters of one node apply to all the nodes of the sub-tree. Such hierarchy creates dependencies between nodes that constrain the graph traversal order. These dependencies make efficient scene graph distribution difficult on a parallel machine [12, 16]. Several scientific visualization tools adopt a data-flow model [8]. An application corresponds to an oriented graph with tasks at vertices and FIFO channels at edges. This graph clearly structures data dependencies between tasks. It eases task distribution on different processing hosts [6].

Hardware limitations have been tackled first by developing graphics supercomputers integrating dedicated hardware [15]. Focus was on increasing the capabilities through different parallelization schemes [14]. Today, such approaches are facing difficulties to keep pace, regarding price and performance, with commodity component based platforms like graphics PC clusters [17]. But aggregating commodity components requires an extra effort on the software side. Chromium [10] proposes a highly optimized streaming protocol, primarily aimed at transporting OpenGL primitives on PC clusters to drive multi display environments. To improve latency, virtual reality oriented libraries duplicate the application on each rendering host. A synchronous broadcast of all input events ensures copies stay coherent [9, 7, 18]. VR applications can also take advantage of a cluster to distribute input devices or simulation tasks. For instance new complex devices, like multi-camera systems [13], increase the number of components to manage and the need for parallel processing. Distributed code coupling have been experimented for VR applications with tools like Covise [6], OpenMask [1] or Avango [20].

All the mentioned algorithms and tools are useful in different application scenarios. Large scale applications often require a number of these technics but it is difficult to choose the most efficient ones and combine them in a single application. In this paper we present a software framework for the development of large distributed VR applications. The goal is to favor the application modularity in an attempt to alleviate software engineering issues while taking advantage of this modularity to enable efficient executions on PC clusters. We developed the FlowVR suite [3, 5], a software suite dedicated to distributed interactive applications.

It is composed of FlowVR (section 2), a middleware that reuses and extends the classical data-flow model, FlowVR Render (section 3), a shader based framework for distributed rendering and VTK FlowVR (section 4) that enables rendering VTK applications with FlowVR Render.

FlowVR comes with a complete set of tools to develop distributed applications, to map an application on a cluster, to launch it and control its execution. FlowVR also comes with tools for graph visualization, trace capture and visualization to analyze an execution.

2 The FlowVR Middleware

2.1 Overview

FlowVR is an open source middleware, currently ported on Linux and Mac OS X platforms. In this section we present its main features. Refer to [3] for more details.

An application is composed of *modules* exchanging data through a *FlowVR network*. A module is usually an existing code that has been updated to call FlowVR functions. A module runs in its own independent process or thread, thus reducing the effort required to turn an existing code into a module.

From the FlowVR point of view, modules are not aware of the existence of other modules. A module only exchanges data with the FlowVR daemon that runs on the same host. The set of daemons running on a PC cluster are in charge of implementing the FlowVR network that connects modules. The daemons take care of moving data between modules using the most efficient method. This approach enables to develop a pool of modules that can next be combined in different applications, without having to recompile the modules.

The FlowVR network defined between modules can implement simple module-to-module connections as well as complex message handling operations. For instance the network can implement synchronizations, data filtering operations, data sampling, dead reckoning, frustum culling, collective communications schemes like broadcasts, etc. This fine control over data handling enables to take advantage of both the specificity of the application and the underlying cluster architecture to optimize the latency and refresh rates.

*email: jean-denis.lesage@imag.fr
†email: bruno.raffin@imag.fr

To execute an application on a cluster the user maps the modules on the different hosts available. The FlowVR network is implemented by a daemon running on each host. A module sends a message on the FlowVR network by allocating a buffer in a shared memory segment managed by the local daemon. If the message has to be forwarded to a module running on the same host, the daemon only forwards a pointer on the message to the destination module that can directly read the message. If the message has to be forwarded to a module running on a distant host, the daemon sends it to the daemon of the distant host. The target daemon retrieves the message, stores it in its shared memory segment and provides a pointer on the message to the receiving module. Using a shared memory enables to reduce data copies for an improved performance.

Daemons can load custom classes (*plugins*) to extend their functionalities. For instance, the current version loads a TCP plugin to implement inter-host communications. Custom plugins can be developed to support other protocols for high performance networks like Infiniband or Myrinet.

2.2 Messages

Each message sent on the FlowVR network is associated with a *list of stamps*. Stamps are lightweight data that identify the message. Some stamps are automatically set by FlowVR. The user can also define new stamps if required. A stamp can be a simple ordering number, the id of the source that generated the message or a more advanced data like a 3D bounding volume. To some extent, stamps enable to perform computations on messages without having to read the message contents. A stamp can be routed separately from its message if the destination does not need it. It enables to improve performance by avoiding useless data transfers on the network.

2.3 Modules

Computation tasks are encapsulated into modules. Each module defines a list of *input ports* and *output ports*. During its execution a module endlessly iterates reading input data from its input ports and writing new results on its output ports. For that purpose it uses the following three main methods:

- The *wait* defines the transition to a new iteration. It is a blocking call that ensures each connected input port holds a new message. Notice that this semantics requires that at each iteration a module receives a new message on each of its connected input ports. This constraint can be loosen by using specific FlowVR network components, as we will see in the following (section 2.4).

- The *get* function enables a module to retrieve the message available on a port.

- The *put* function enables a module to write a message on an output port. Only one new message can be written per port and iteration. This is a non-blocking call, thus allowing to execute computations and communications in parallel.

Each module has two predefined ports called *beginIt* and *endIt*. The *input activation port beginIt* is used to lock the module to an external event. The *output activation port endIt* is used to signal other components that the module has started a new iteration.

A module does not explicitly address any other FlowVR component. Its only exchange channel with the outside FlowVR world is through its ports. This ensures modules can be reused in different applications without code modification or recompilation.

Usually a module is build using an existing piece of code that is modified to include the required FlowVR function calls. It runs in its own process or thread as it would before becoming a module. A module can be programmed in any language as long as the FlowVR library provides the required language binding. The current implementation only provides a C++ binding. Other languages will be supported in the future.

2.4 The FlowVR Network

The FlowVR network specifies how the ports of the modules are connected. The simplest primitive used to build a FlowVR network is a *connection*. A connection is a FIFO channel with one source and one destination.

To perform high performance and complex message handling tasks we introduce a new network component called *filter*. Like a module, a filter is a computation task that has typed ports. But filters are deeply different from modules in two different ways:

- A filter is not constrained to receive one and only one message per input port and per iteration. A filter has access to the full list of incoming messages. It has the freedom to select, combine or discard the ones it wants. It can also create new messages. For instance, a filter can discard incoming messages which 3D bounding box falls outside of a given volume.

- A filter does not run in its own process. It is a plugin loaded by FlowVR daemons. The goal is to favor the performance by limiting the required number of context switches.

As such, a filter is more difficult to program than a module regarding message handling. Usually, a user only selects the filters it needs amongst the ones that come with FlowVR.

Amongst filters, we call *routing nodes* the filters that simply forward all incoming messages on one or several outputs. They are useful to set custom routing graphs.

We also distinguish another special class of filters, called *synchronizers*. A synchronizer implements coupling policies by centralizing data from other filters or modules to take a decision that will then be executed by other filters. A synchronizer differs from standard filters because all input and output connections only carry the message stamps alone.

Assembling synchronizers and filters can lead to complex data distribution strategies. For instance, a synchronizer is able to limit frequency of a component or to synchronize iterations from two components. Some networks are able to change the default FIFO connection strategy and enable to couple components running at different frequencies.

Each FlowVR application is managed by one special module called a *controller*. The controller first starts the application's modules using the launching command computed by *flowvr-module*. Once modules are launched, they register themselves to their local daemon which sends an acknowledgment to the controller. Then, the controller forwards a description of FlowVR network to the daemons. They create this network by configuring themselves (load plugins, set parameters, etc.). The execution of the application can then start.

2.5 Hierarchical components

FlowVR includes a C++ library dedicated to the design of large networks. This library implements a hierarchal components model [11]. Hierarchy eases modularity and reusability of FlowVR networks. Two kinds of components are defined in this model:

- *Primitive* component is a basic FlowVR object (ie module, filter, connection, synchronizer or routing node).

- *Composite* component contains other components (primitive or composite). Links connect ports between different components. These links can connect two components of the same level of hierarchy or connect a composite to one of his children components. Links can not cross a component hull, encapsulation is strict.

A library of composite components for the development of large parallel applications is provided with FlowVR. It contains:

- *Parallel patterns.* They ease the interface with parallel modules implemented with threads, MPI or other parallel middlewares. From the user's view, component encapsulation hides some parallelism issues such as communications between parallel processes for instance.

- *Communication patterns.* They implement efficient communication schemes between parallel components.

- *Synchronization patterns.* They are used with a FlowVR synchronizer to sample messages along a connection. They enable to couple two components that runs at different frequencies.

- *High level components.* They are usually implemented by the FlowVR community. They encapsulate a part of a FlowVR network that can be used directly in different applications. Some of them make the interface with external libraries such as VRPN [19] (VR devices library) or SOFA [2] (physical simulation library).

3 FlowVR Render

FlowVR Render [5] is a shader based parallel rendering framework relying on FlowVR. It takes advantage of the power offered by graphics clusters to drive display walls or immersive multi-projector environments like Caves. It defines graphics primitives using shader programs to propose a high performance communication protocol:

- Shaders are used to specify the visual appearance of graphics objects. They require only a few parameters and not the full complexity of the fixed-function OpenGL state machine. It leads to a simpler protocol that does not have to manage state tracking. Those primitives are self-contained.

- Shaders enables to easily take advantage of all features offered by programmable graphics cards.

- FlowVR Render works in retained-mode. Only updates of primitives need to be sent.

The rendering framework is based on several *viewers* creating the scene and distributed *renderers* rendering the scene. A *viewer* describes *primitives* sent to a *renderer* using the FlowVR Render protocol. The *renderer* is in charge of rendering this set of primitives. All of them are FlowVR modules.

We developed a wrapper that reads back the image computed by an OpengGL application, turn it into a FlowVR Render primitive using the image as a texture. It enables to render unmodified OpenGL applications on multi display environments with FlowVR Render.

FlowVR Mplayer is a port of the MPlayer Movie Player that uses FlowVR Render. This enables to play movies on multi display environments. It aims at taking advantage of the high resolution of screen walls, and allows to play high resolution videos.

4 VTK-FlowVR

VTK FlowVR enables to perform VTK data visualization using FlowVR Render with minimal modifications of the original code. It also enables to encapsulate VTK code into FlowVR modules to get access to the FlowVR capabilities for modularizing and distributing VTK processings.

5 Making Development Easier

FlowVR provides tools to ease the development or the debugging of an application:

- FlowVR-GLGraph, an OpenGL based FlowVR network viewer, gives users the opportunity to display the network of the instanciated application. It features color handling, zoom, hiding of uninteresting parts to make the network more readable (Fig. 1).

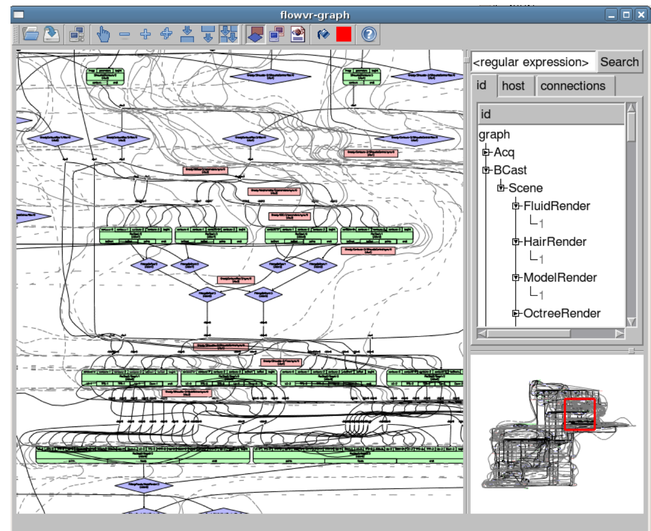

Figure 1: FlowVR OpenGL graph viewer

- FlowVR-GLTrace is a trace visualization tool. FlowVR supports capture of predefined or user-defined events. Capture is performed with minimal impact on the execution performance. Once stored on disk, FlowVR-GLTrace enable to process and display the execution trace. This tool helps debugging by showing the user the chronology of events, the messages exchanged between the various components involved in an application.

6 Application Example

One large VR application developed with FlowVR aggregates:

- a parallel real-time multi-camera 3D modeling algorithm directly implemented with FlowVR

- SOFA [2], a physical simulation library. SOFA is not developed with FlowVR, but it has been encapsulated into a FlowVR component

- distributed rendering implemented with FlowVR-Render.

It runs on a PC cluster, some nodes of the cluster being dedicated to computations, some nodes are attached to video cameras and the others drive displays or video-projectors. A light version of this application has been demonstrated at Siggraph Emerging Technologies 2007 [4]. An user puts his hands in a reconstruction area filmed by cameras (figure 2). The images from the different view points are processed online to provide a 3D model of the user's hands. This 3D model is sent to SOFA [2] to compute in real time interactions between the 3D model and the pure virtual objects. The scene is them rendered on displays, after texturing the 3D model with photometric data extracted from the camera images ([1]).

[1] videos available at http://www.inrialpes.grimage.fr

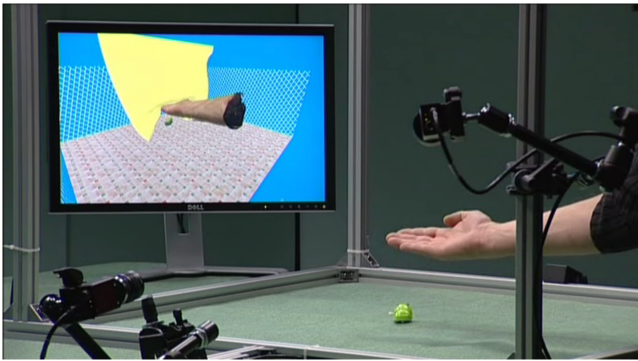

Figure 2: GrImage: interactions with soft virtual objects

7 Conclusion

We presented the FlowVR suite. It provides a middleware for high performance interactive computing favoring code reuse and modularity. On top of this middelware FlowVR Render defines a communication protocol for graphics primitives. It enables an efficient remote rendering on multi display environments. The suite is complemented by a video player based on Mplayer, an OpenGL wrapper and components for coupling VTK, FlowVR and FlowVR render.

Acknowledgment

The main contributors to FlowVR and its components are Jrmie Allard, Jean-Denis Lesage, Clment Mnier, Sabastien Limet, Emmanuel Melin, Bruno Raffin and Sophie Robert.

This work was partly funded by Agence Nationale de la Recherche contract ANR-06-MDCA-003.

References

[1] OpenMASK. http://www.irisa.fr/siames/OpenMASK.

[2] J. Allard, S. Cotin, F. Faure, P.-J. Bensoussan, F. Poyer, C. Duriez, H. Delingette, and L. Grisoni. SOFA: an Open Source Framework for Medical Simulation. In *Medicine Meets Virtual Reality (MMVR)*, 2007.

[3] J. Allard, V. Gouranton, L. Lecointre, S. Limet, E. Melin, B. Raffin, and S. Robert. FlowVR: a middleware for large scale virtual reality applications. In *Euro-Par 2004 Parallel Processing: 10th International Euro-Par Conference*, pages 497–505, Pisa, Italia, August 2004.

[4] J. Allard, C. Ménier, B. Raffin, E. Boyer, and F. Faure. Grimage: Markerless 3D Interactions. In *Proceedings of ACM SIGGRAPH 07*.

[5] J. Allard and B. Raffin. A shader-based parallel rendering framework. In *IEEE Visualization Conference*, Minneapolis, USA, October 2005.

[6] A.Wierse, U.Lang, and R. Rhle. Architectures of Distributed Visualization Systems and their Enhancements. In *Eurographics Workshop on Visualization in Scientific Computing*, Abingdon, 1993.

[7] A. Bierbaum, C. Just, P. Hartling, K. Meinert, A. Baker, and C. Cruz-Neira. VR Juggler: A Virtual Platform for Virtual Reality Application Development. In *IEEE VR 2001*, Yokohama, Japan, March 2001.

[8] K. W. Brodlie, D. A. Duce, J. R. Gallop, J. P. R. B. Walton, and J. D. Wood. Distributed and collaborative visualization. *Computer Graphics Forum*, 23(2), 2004.

[9] C. Cruz-Neira, D. J. Sandin, T. A. DeFanti, R. V. Kenyon, and J. C. Hart. The Cave Audio VIsual Experience Automatic Virtual Environement. *Communication of the ACM*, 35(6):64–72, 1992.

[10] G. Humphreys, M. Houston, R. Ng, S. Ahern, R. Frank, P. Kirchner, and J. T. Klosowski. Chromium: A Stream Processing Framework for Interactive Graphics on Clusters of Workstations. In *Proceedings of ACM SIGGRAPH 02*, pages 693–702, 2002.

[11] J.-D. Lesage and B. Raffin. A Hierarchical Programming Model for Large Parallel Interactive Applications. In *IFIP International Conference on Network and Parallel Computing*, pages 516–525, 2007.

[12] B. MacIntyre and S. Feiner. A distributed 3D graphics library. In M. Cohen, editor, *Proceedings of ACM SIGGRAPH 98*, pages 361–370. Addison Wesley, 1998.

[13] W. Matusik and H. Pfister. 3D TV: A Scalable System for Real-Time Acquisition, Transmission, and Autostereoscopic Display of Dynamic Scenes. In *Proceedings of ACM SIGGRAPH 04*, 2004.

[14] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics and Applications*, 14(4):23–32, July 1994.

[15] J. S. Montrym, D. R. Baum, D. L. Dignam, and C. J. Migdal. InfiniteReality : A Real-Time Graphics System. In *Proceedings of ACM SIGGRAPH 97*, pages 293–302, Los Angeles, USA, August 1997.

[16] M. Roth, G. Voss, and D. Reiners. Multi-threading and clustering for scene graph systems. *Computers & Graphics*, 28(1):63–66, 2004.

[17] R. Samanta, T. Funkhouser, K. Li, and J. P. Singh. Hybrid sort-first and sort-last parallel rendering with a cluster of PCs. In *ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, August 2000.

[18] B. Schaeffer and C. Goudeseune. Syzygy: Native PC Cluster VR. In *IEEE VR Conference*, 2003.

[19] R. M. Taylor II, T. C. Hudson, A. Seeger, H. Weber, J. Juliano, and A. T. Helser. VRPN: a device-independent, network-transparent VR peripheral system. In *ACM Symposium on Virtual Reality Software & Technology 2001*, 2001.

[20] H. Tramberend. Avocado: A distributed virtual reality framework. In P. A. L. Rosenblum and D. Teichmann, editors, *Proceedings IEEE Virtual Reality 99 ConferencE*, pages 14–21, March 1999.