HAL
open science

Modularity for Large Virtual Reality Applications

Jérémie Allard, Jean-Denis Lesage, Bruno Raffin

To cite this version:

Jérémie Allard, Jean-Denis Lesage, Bruno Raffin. Modularity for Large Virtual Reality Applications. Presence: Teleoperators and Virtual Environments, 2010, 19 (2), pp.142-161. 10.1162/pres.19.2.142 . hal-00688474

HAL Id: hal-00688474

https://inria.hal.science/hal-00688474v1

Submitted on 17 Apr 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Modularity for Large Virtual Reality Applications

Jérémie Allard[1,2] Jean-Denis Lesage[1,3]
Bruno Raffin[1,3]
[1] INRIA
[2] Laboratoire d'Informatique Fondamentale de Lille (LIFL)
[3] Laboratoire d'Informatique de Grenoble (LIG)

Abstract. This paper focuses on the design of high performance VR applications. These applications usually involve various I/O devices and complex simulations. A parallel architecture or grid infrastructure is required to provide the necessary I/O and processing capabilities. Developing such applications faces several difficulties, two important ones being software engineering and performance issues. We argue that application modularity is a key concept to help the developer handle the complexity of these applications. We discuss how various approaches borrowed form other existing works can be combined to significantly improve the modularity of VR applications. This led to the development of the FlowVR middleware that associates a data-flow model with a hierarchical component model. Different case studies are presented to discuss the benefits of the approach proposed.

1 Introduction

Virtual Reality encompasses various types of applications that may require different middleware libraries for fast prototyping, distributed VR environments or web based VR for instance. We focus on high performance virtual reality applications. They associate multiple input and output devices as well as significant computing resources to enable advanced interactions with computationally and data intensive simulations. Their computing and I/O needs require executions on parallel machines or grid infrastructures. When developing such application the main difficulties to overcome include:

- Algorithmic issues to run correct simulations, to produce convincing and useful images or other non-visual renderings, to integrate and extract data from various sensors like cameras.
- Software engineering issues where multiple pieces of codes (simulation codes, graphics rendering 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 performance limitations bypassed by multiplying the units available (disks, CPUs, GPUs, cameras, video projectors, etc.), but introducing at the same time extra complexity. In particular it often requires introducing parallel algorithms and data redistribution strategies that should be generic enough to minimize human intervention when the target execution platform changes.

In this paper, we present the design of the FlowVR middleware. FlowVR is an open source middleware library. Its development followed our former experience with Net Juggler (Allard, Gouranton, Lecointre, Melin, & Raffin, 2002), an early extension of the VR Juggler library for PC cluster support (Bierbaum et al., 2001), and various attempts to couple VR and parallel simulations.

FlowVR is designed with the goal of enforcing a modular programming that leverages software engineering issues while enabling high performance executions on parallel architectures. We targeted a middleware tailored for high performance interactive applications rather than a generic purpose one. Classical parallel programming environments first focus on performance, giving users a fine control on parallelism, but often impairing modularity, making the maintenance of large heterogeneous applications difficult. The base concepts FlowVR is built upon are:

- A static data-flow model. An application is seen as a static graph, where vertices are computing tasks distributed on computing resources and edges are FIFO data communication channels.
- A task, or component, executes an endless iterative loop ignoring networking issues. It repeatedly gets data from input ports, process the data and provide results on output ports.
- A set of tasks can actually be a parallel application, i.e. tasks that communicate between each-other by other means than the data channels defined by the graph. The middleware should not be aware of these communications and should not impair them.
- A hierarchical component model encapsulating arbitrarily complex patterns in components to favor modularity and code reuse.
- Data exchange schemes that are external to computation tasks. The hierarchical component model enables to design advanced communication schemes like collective communications, data sampling policies and complex synchronization patterns.
- A component programming interface designed to lead to minimal code modifications when turning an existing code, parallel or not, into a component.
- An application preprocessing step to specialize the application for a given target architecture. It favors application portability.

After discussing related approaches (section 2), the paper focuses on the various concepts we harnessed to enable a high level modularity, the cornerstone of the FlowVR design (section 3). We report our experience developing some large applications (section 4), discuss the main FlowVR limitations we experienced (section 5) before to conclude (section 6).

2 Related Work

There are multiple environments that support interactive applications: from the virtual applications running on simple mobile phone to the distributed virtual environments like *Second Life*. Meanwhile, the virtual reality community is also very heterogeneous,

containing for example artists, cognitive scientists or expert engineers. In response, a wide collection of virtual reality toolkits has been developed to satisfy all these needs.

Some toolkits like Virtools, HECTOR (Wetzstein et al., 2007), Avango (Kuck, Wind, Riege, & Bogen, 2008) or InTml (Figueroa, Bischof, Boulanger, Hoover, & Taylor, 2008) provide a set of abstractions that eases the development of applications. For example, it can be high level scripting languages, efficient editing or visualizing tools for complex data as 3D objects. These middlewares can be used for prototyping applications. They are also the appropriate tools for non-expert developers to create VR applications. For large applications, these tools may reach their limits. A point-and-click interface may be tedious and repetitive when applying the same operation to thousands of objects. In this case, the possibility to access a low-level language to automate these tasks is crucial and more reliable.

Several VR frameworks support some level of parallelism for multi-projector rendering and distributed input devices (Soares, Raffin, & Jorge, 2008). For instance VR juggler (Bierbaum, Hartling, Morillo, & Cruz-Neira, 2005) uses CORBA for distributing input devices. Parallel simulations coupling is supported in some cases relying on simple data redistribution mechanisms (Wierse, Lang, & Rühle, 1993).

The data-flow model is classic for scientific visualization middlewares. Some of them support applications involving parallel code as MPI processes. But usually distributed executions and parallel code coupling are not first class features and their support is limited (Brodlie, Duce, Gallop, Walton, & Wood, 2004). Data transfer can be managed in different ways. The two main policies are the push (data-driven) or pull model (demand-driven). For instance, VTK (Ahrens, Law, Schroeder, Martin, & Papka, 2000) and VISIT (Childs et al., 2005) are both pull based. OpenDX (Abram & Treinish, 1995) and Iris Explorer (Foulser, 1995) use a push one. Data-flow model has been adopted by other communities and tends to replace the client-server paradigm to improve scalability (Lohse, Winter, Repplinger, & Slusallek, 2008).

In a distributed context the pull model can impair performance, as it adds the latency caused by the request message. For this reason FlowVR adopts a push model. But because it allows cycles, it can also implement pull actions. It enables to mix between a push or pull model depending on the application requirements.

OpenMask (Margery, Arnaldi, Chauffaut, Donikian, & Duval, 2002) differs from the classic push or pull models. It makes sampling one of its base concept. OpenMask is based on a data-flow model where each task run asynchronously. Each data stream is associated with an interpolation/extrapolation function to resample it according to the destination data needs. FlowVR filters can be used to implement similar resampling schemes.

Component oriented middleware libraries like CORBA were first developed for distributed applications, with flexibility as their primary goal. They usually do not integrate abstractions for parallel programming, have performance issues, and often require substantial modifications of existing applications. This results in tools that are not well adapted for high performance computing, motivating the development of extensions and new models (Keahey & Gannon, 1997; Denis, Pérez, & Priol, 2003; Beckman, Fasel, Humphrey, & Mniszewski, 1998; Armstrong et al., 1999). Notice in particular the on-going project SCIRun2, whose goal is to extend the SCIRun environment to support the CCA component model for high performance computing (K. Zhang, Damevski, Venkatachalapathy, & Parker, 2004).

The component models aforementioned, mainly derived from CCM, do not support component hierarchy. It is however one of the base features of Fractal (Bruneton, Coupaye,

Leclercq, Quéma, & Stefani, 2006). But like the CCM model, parallel code coupling is not directly supported. Pro-active, a grid middleware, implements part of the Fractal specification and supports both, component hierarchy and parallel code coupling (Baude, Caromel, & Morel, 2003). However redistribution patterns are coded into the ports of the parallel components. A pattern cannot be modified without affecting the component, limiting the application modularity.

FlowVR adopts a component oriented approach based on a data-flow model rather than remote procedure calls. It takes advantage of its static data-flow model to build component hierarchies, while enabling parallel component coupling. One important issue of parallel code coupling is data redistribution (classically called the $N \times M$ problem) (L. Zhang & Parashar, 2006; Richart, Esnard, & Coulaud, 2007). The hierarchical FlowVR model enables to encapsulate redistribution patterns into components that are independent from the data producers and consumers. Though the model should support complex $N \times M$ redistribution patterns, we have not yet implemented such components.

The skeleton model proposes a pattern language for parallel programming (Cole, 1989; Mattson, Sanders, & Massingill, 2004). A program is written from the composition of predefined parallel patterns. Various environments rely on this model like ASSIST (Aldinucci, Coppola, Danelutto, Vanneschi, & Zoccolo, 2006) for grid computing or Skipper-D (Serot & Ginhac, 2002) for vision applications. Skeletons have a clear semantics, can be associated to a cost model and hide their implementation details to the application developer. Given the target architecture, the application is compiled down to a specialized parallel code. Hierarchies of skeletons are supported by some environments like Skipper-D.

FlowVR composite components can be considered as parametric skeletons. Jubertie *et al.* propose in (Jubertie, Melin, Vautard, & Lallouet, 2008) a cost model for the data-flow graph that could be extended to composite components. However, FlowVR components tend to be closer to design patterns than skeletons. Application developers can freely ignore all predefined patterns and develop their own patterns. There is no discontinuity between primitive components and high level ones that would make such development specifically difficult.

3 The FlowVR Model

In this section, we present the basic concepts of FlowVR. First, we describe the execution model (Allard et al., 2004) and the run-time environment. Next, we present the application description language based on a hierarchical component model (Lesage & Raffin, 2008).

A classical VR application tends to be monolithic in the sense that it is built around a single loop that:

- updates the input states,
- updates the virtual scene,
- publishes the new state of the scene for various renderings (image, sound, force feedback).

This sequence imposes the different steps to be synchronous, i.e. to have the same refresh rate. Of course, it is possible to circumvent this loop. This is classically the case for instance when reading the state of input devices. Each input device is usually managed in an external thread or process. It asynchronously updates a buffer that the application

reads when needed. A double buffering mechanism ensures a safe concurrent read/write access. The main loop stays anyway the central coordination point of the application. It impairs the ability to design and execute an application taking advantage of numerous distributed resources. We switched to a different approach, borrowed from scientific visualization tools like Iris Explorer (Foulser, 1995). The application is seen as a set of tasks, each one executing a local loop, synchronized through the flow of data.

3.1 Execution Model

A FlowVR application is a set of distributed iterative tasks. These tasks, also called components, ignore networking issues. They simply get data from input ports, process the data and provide results on output ports. They are interconnected by FIFO data communication channels shaping the network graph. At execution these tasks are distributed on the target machines and the run-time environment takes care of moving data between tasks.

3.1.1 Module

The base component of FlowVR is the *module*. A module has input and output ports. It executes an endless loop and its application programming interface (API) is built around three basic instructions:

- *wait*: lock the module as long as no new message is available on each of its connected input port.
- *get*: get a pointer on the new message received on a given port.
- *put*: publish a new message on a given output port.

There is no imposed form for a module as long as its semantics is respected. It can be a process, a thread, a group of threads collaborating to implement a module. The API is minimal and simple to ease turning existing codes, being multi-threaded, parallel or even GPU kernels, into modules.

Similarly to classical parallel programming environments, the FlowVR API only provides low level message handling functions. A message is simply a sequence of bytes a module has free access to. This enables a fine control on data copies that can induce significant costs for large messages. Higher level message handling methods can be implemented using this API, but they are not part of the core FlowVR software.

By default each module has:

- an output port *endIt* where a message is sent every time the module enters the *wait* instruction,
- an input port *beginIt* that, when active, locks the module as long as no message is received on the port (the message is used as an event and its content is ignored).

3.1.2 Filter

A *filter* has input and output ports. In opposition to modules that can only access the last message received on each input port, a filter has access to the full buffer of incoming messages stored locally. It can freely modify or discard any of these messages. A filter

can for instance resample messages, by discarding all incoming messages except the last one that it forwards on its output port.

The API to develop modules is intentionally simple and constraining (see the *wait* semantic for instance). The goal is to keep module development simple. In opposite, filters offer more freedom in particular regarding buffer access. Usually an application developer does not have to develop new filters. Filters provided with FlowVR perform generic message handling tasks, making them easy to reuse in multiple applications. Combining filters and modules enable to implement complex behaviors as we will see in the following section.

3.1.3 Connection

A *connection* is a simple FIFO channel connecting an input port to an output port. Several connections can connect to one output port. In this case each message available on the output port is broadcasted on each connection. One input port can only have one incoming connection.

The simplest application a user can write involves two modules connected through a connection. The size of the buffer associated to a connection is only limited by the amount of memory available. If the receiver is slower than the sender, an overflow will occur once the memory is saturated. We will see later how to avoid such situation.

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 content. The stamp list can be sent on the network without the message payload if the destination does not need it. It improves performance by avoiding useless data transfers.

3.2 Example: A Simple Visualization Application

We detail a simple application to show the modularity enabled by this model. This application connects one data producer, the module *compute*, and the *visu* consumer module (Figure 1a). This could be a 3D mesh generator linked to an OpenGL rendering process. We show how we can change the way data are exchanged between both modules simply by changing the network.

3.2.1 Data-Driven Policy (Figure 1a)

The simplest way to link the two components is to use a single direct connection. In this case the consumer frequency reaches at most the one of the producer. If the consumer is slower than the producer, the number of messages sent will grow because the consumer will not be able to process them. In the model, the buffers associated to FIFO connections are unlimited, but practically they are limited by the amount of memory available. So a memory overflow can occur.

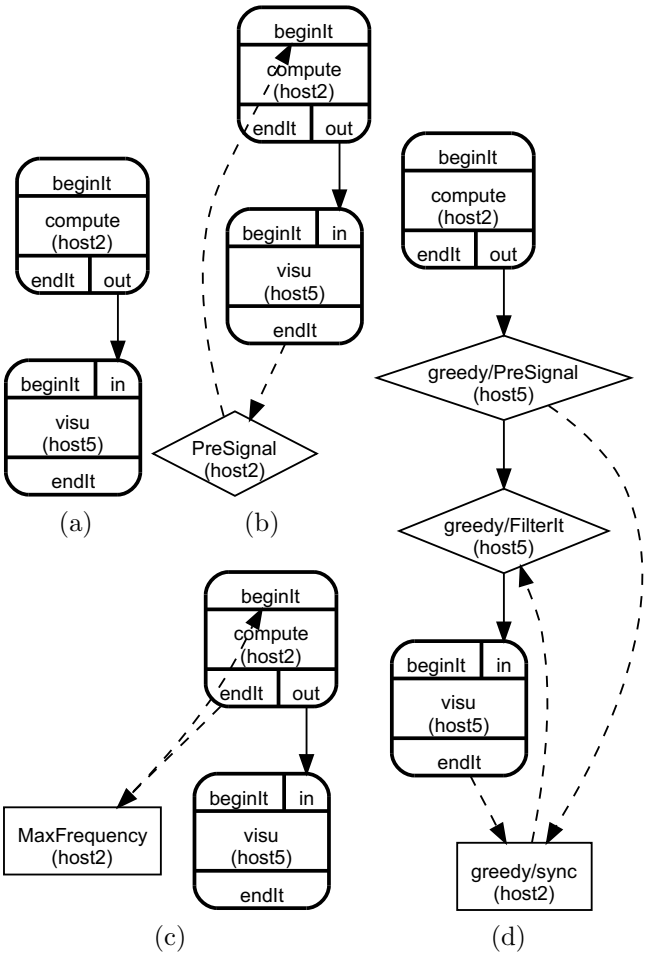


Figure 1: *[one column width]* (a) A simple FlowVR application with only one connection between two modules. (b) The *visu* module pull on demand messages from *compute*. (c) The max frequency filter bounds the frequency of the *compute* module. (d) The greedy pattern enables *compute* and *visu* to run at independent frequencies. Dashed arrows represent connections where only the message's stamps are transferred.

3.2.2 Demand-Driven Policy (Figure 1b)

A simple approach to avoid such overflow is to switch from a push to a pull paradigm. Because FlowVR allows cycles in the data-flow graph, it is possible to have the *visu* module controlling the frequency of *compute*. Each time *visu* ends an iteration, it sends a request message on its *endIt* port. This message is forwarded through a *PreSignal* filter to the *beginIt* port of *compute*. This message will unlock *compute*, that will proceed to produce a new data. The *PreSignal* filter is required to avoid the deadlock caused by the cycle. This filter sends a first message before forwarding incoming messages : it puts an initial token into the cycle to unlock it.

3.2.3 Data-Driven Policy with Frequency Constraint (Figure 1c)

To control the frequency of a module, another approach is to use a *MaxFrequency* filter. Each time the module ends an iteration, it sends a request message to this filter (*endIt* port). From these messages, the filter is able to compute the frequency of the *compute* module. The filter can control the module frequency by sending messages to the *beginIt* port up to a limit custom maximum frequency.

Notice that here we have a cycle too. In this case the *MaxFrequency* filter is in charge of sending the unlocking first message. As this filter is always used in cycles, it is convenient to give it this responsibility.

3.2.4 Asynchronism Based on Resampling (Figure 1d)

For large interactive applications, having all modules running at the frequency of the slowest one can severely affect the reactivity of computations. A possible approach is to consider that a data stream is a sampled signal that can be resampled. The consumer only gets the data it can process, discarding the other ones.

We present the greedy pattern, a basic resampling pattern favoring the reactivity by providing the consumer with the most recent data available (Figure 1d). Because filters have access to the full buffer of messages stored locally, it can easily implement a sampling policy. This pattern is organized around a special synchronization filter, called a synchronizer, and 2 filters.

Each time the module *visu* ends an iteration, the synchronizer receives one message. The synchronizer also receives a stamp message for each message sent by the producer. This message is actually provided by the *PreSignal* filter to unlock the cycle created by the synchronizer. When the synchronizer receives a request from *visu*, it forwards to the *FilterIt* filter the stamp of the last stamp message received from *PreSignal*. *FilterIt* waits to receive the full message having this same stamp, discards all older messages locally stored and forwards this message to *visu*. If no message is available when the synchronizer receives the request, it tells the module to reuse the last message already received, sending it a special empty message.

FlowVR enables to separate task implementation from their coupling. For all these simple examples, the modules did not have to be recompiled. Only the network specification changed. It enables to easily test various network options. Following the same principle, collective communications between parallel tasks can be implemented through different combinations of filters and connections. For instance a gather collective communication can be built from a tree of filters that merges two incoming data streams into a new one.

Building large applications can lead to complex graphs. Relying on a hierarchical

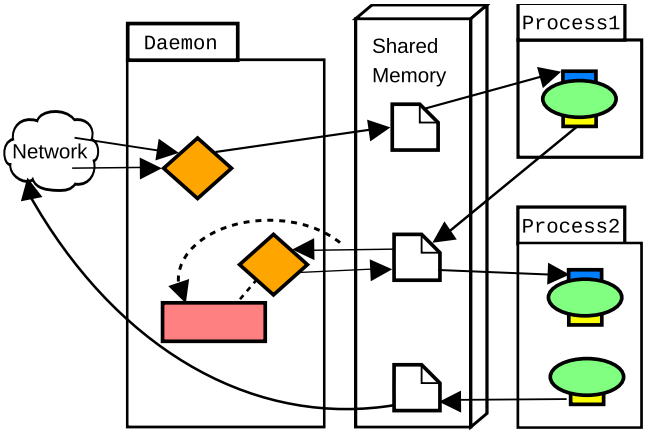


Figure 2: *[one column width]* The FlowVR daemon acts as a broker between modules (green ellipses), using filters (orange diamonds) for resampling and simple data processing, and synchronizers for synchronization policies (pink rectangle).

component enables to cope with this complexity and to keep a high modularity level. The hierarchy can also be exploited by tools to ease the development of large interactive applications (Figure 3). But before to detail the application description environment, let focus on the run-time environment that is in charge of the execution.

3.3 Run-time Environment

FlowVR run-time environment relies on a daemon that runs on each host of the target machine. This daemon is in charge of:

- registering each module running on its host,
- implementing all message exchanges transiting through its host,
- executing all local filters.

Modules are launched directly using their own command. FlowVR does not imposes any specific launching mechanism to ease portability of existing codes. For instance a MPI code can use *mpirun*. Once launched, each module registers at the daemon of its host.

The daemon manages a shared memory segment used to store all messages it handles (Figure 2). Local modules and filters have a direct access to this memory under the control of the daemon that ensures memory allocation, safe read and write accesses and garbage collection. Having direct access to the shared memory segment saves data copies.

When a module executes a *put*, it tells the daemon that the message is ready to be forwarded to its destination. If the destination module runs on the same host, the demon provides a pointer to the module that directly reads the message from the shared memory. If the message has to be forwarded to a distant module, 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 to the destination module, which processes it exactly as for local communications.

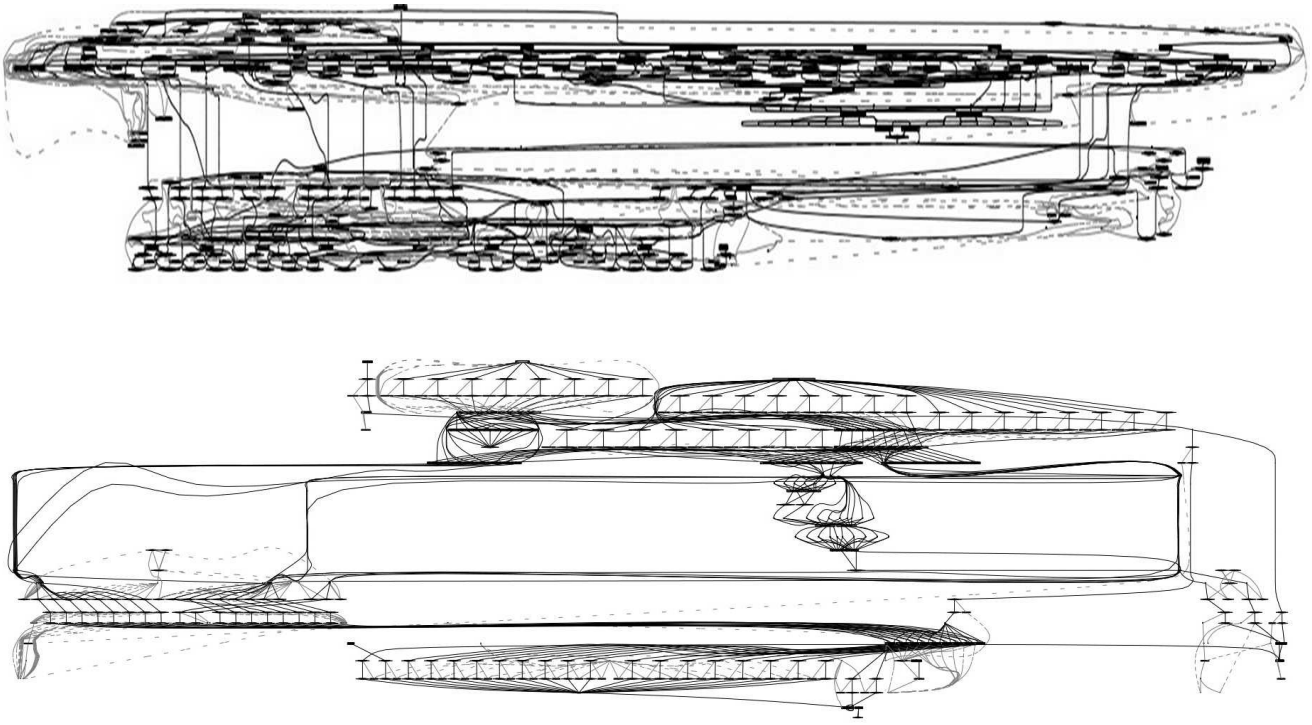


Figure 3: *[two columns width]* Two representations of the same data-flow graph of an application running on the Grimage platform. Each edge represents a connection and each vertex a module or a filter. The first representation (top) does not use the hierarchical structure. The layout algorithm used to produce the second representation (bottom) relies on the hierarchy. Some logical patterns embedded in composite patterns become visible.

Each daemon stores a table of actions to perform on messages according to their origin. An action can be a simple message routing, or the execution of a filter. Filters are plugins loaded by the daemons when starting the application. Here, an important difference between a filter and module appears. A filter is necessarily a daemon plugin, while a module is an external process or thread. This design is required to provides the best flexibility for programming modules, while providing minimum overhead in complex filtering networks.

The daemon and its plugins are multi-threaded, ensuring a better performance scalability on multi-core and multi CPU architectures.

3.4 Hierarchical Component Model

Because FlowVR is designed for large applications, the data-flow graph can be complex (Figure 3). We have to provide the user tools to face this complexity and avoid the burden of explicitly describing such a graph. We rely on the composite design pattern (Gamma, Helm, Johnson, & Vlissides, 1995) to support hierarchies of components. This enables to encapsulate in one component a complex pattern recursively built from simpler ones. The application is processed by multiples *traverses* of this hierarchical description to extract the data-flow graph and all data required to run the application (Figure 4). Each traverse calls on each component a *controller*, a specific function in charge of a component given aspect.

A component defines input and output ports. We distinguish two kinds of components:

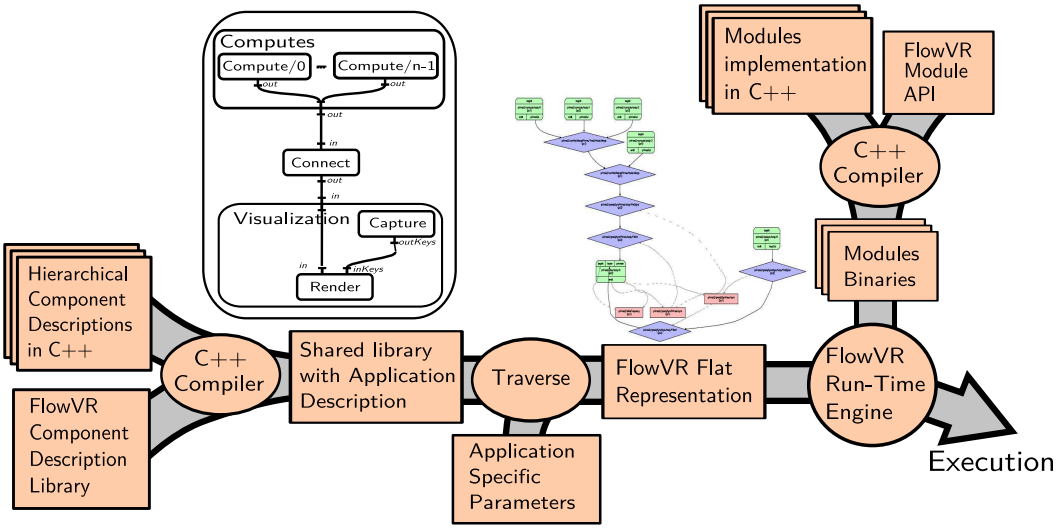


Figure 4: *[two columns width]* The FlowVR front-end. Components (left to right) are compiled, loaded and traversed to provide the module launching commands and the instruction sets for the daemons. Once compiled, modules (top to bottom) are started as defined by the application.

Primitive components. A primitive component is a base component that cannot contain other components. Primitive components are modules, filters and connections.

Composite components. A composite component contains other components (composite or primitive). Each port is visible from the outside and the inside of the component. Component encapsulation is strict. A component can not be directly contained into two parent components.

A link connects two component ports. It cannot directly cross a component membrane. A link between 2 ports is allowed only for the 2 following cases:

- A descendant link connects a port of a parent composite component to a port of one of its child component. Such links must always connect an input/input or output/output pair of ports.
- A sibling link connects two ports of two components having the same parent component. Such a link must always connect an input/output pair of ports.

A simple and common composite component is a *metamodule*. A metamodule handles modules that are logically related, in particular when they are all launched from a single command. This is for instance the case for a MPI code that uses `mpirun` to start all its processes. Such a metamodule takes as parameter the list of hosts where to start the program. From this list of hosts it defines the different modules, one per MPI process, and the exact syntax of the starting command.

In this model, a FlowVR application becomes actually one component. Each component is defined locally according to the other sibling components and its parent. It favors modularity. The developer does not need to control all components of the application as he would if he had to directly write the data-flow graph. For instance, all components need to have a unique id. With the hierarchical model, the user just needs to make sure the component he develops has an id different from the other sibling components having the same parent component. He does not have to pay attention to other component ids.

Then, the components of the data-flow graph are produced by processing the hierarchical description. If we consider the examples of the component ids, the data-flow graph component ids are obtained by concatenating all ascendants ids. The uniqueness of ids between siblings components ensures the final ids are all unique.

Application processing, to detect configuration errors and produce the data-flow graph for instance, is performed through a sequence of *traverses* that call *controllers* on components. A controller is local to a component. It can only modify the state of its component. It can read the state of other components its owner is linked to (directly or not). A component can have several controllers. It usually enforces modularity to have multiple specialize controllers. We distinguish introspection controllers that just get data from its component. For instance, this is an introspection controller that is executed to concatenate ids. We also have configuration controllers that modify their component state. For instance the modules of a MPI metamodule are produced by such controller once the number of processes to launch is known. The execution order of various controllers becomes significant once at least one configuration controller is called.

Because components are developed using object inheritance, controllers are overwritten only when required, the inherited controller being used otherwise.

One *traverse* just calls one controller per component. During a traverse, parameters can be exchanged between controllers to propagate data. It enables for instance to get the concatenation of all ascendant names and a file where to append the newly computed component name.

The FlowVR application processing apply a sequence of 4 controllers:

- An *execute* configuration controller creates child components and ports according to their environment (external parameters as well as other component states). For a MPI metamodule it creates the list of modules from the lists of hosts where to execute the application.

- A *mapping* configuration controller defines the host each primitive component is executed on. For a MPI metamodule, it associates one host name to each child module.

- A *run* introspection controller extracts the launching command for each metamodule.

- An *XMLBuild* introspection controller builds the data-flow graph of the application (using the XML markup language).

Here is an example of a component hierarchy (Figure 5). The *Computes* metamodule is a MPI application spawning several modules. The *Visualization* metamodule only spawns two modules (interleaved threads), one dedicated to keyboard and mouse event capture, and the other to process and render incoming data. To interconnect both components, we use an extra composite component, called *Connect*. *Connect* is in charge of gathering the partial results from the various *Compute/i* processes to forward a single message containing a full simulation state to *Visualization*. *Connect* is built from the *NtoOne* component. This component encapsulates a generic tree pattern for data redistribution. *Connect* just sets the parameters of *NtoOne*: the arity of the tree (2) and the type of the component used for the tree nodes (*Merge*). The actual content of *NtoOne* is known once *Computes* is properly instantiated. Only at this point *NtoOne* knows how many pieces of data it has to gather to set the tree depth. The *execute* controller of *NtoOne* must be executed after the one of *Computes* that creates the modules *Compute/i*. If *Computes*

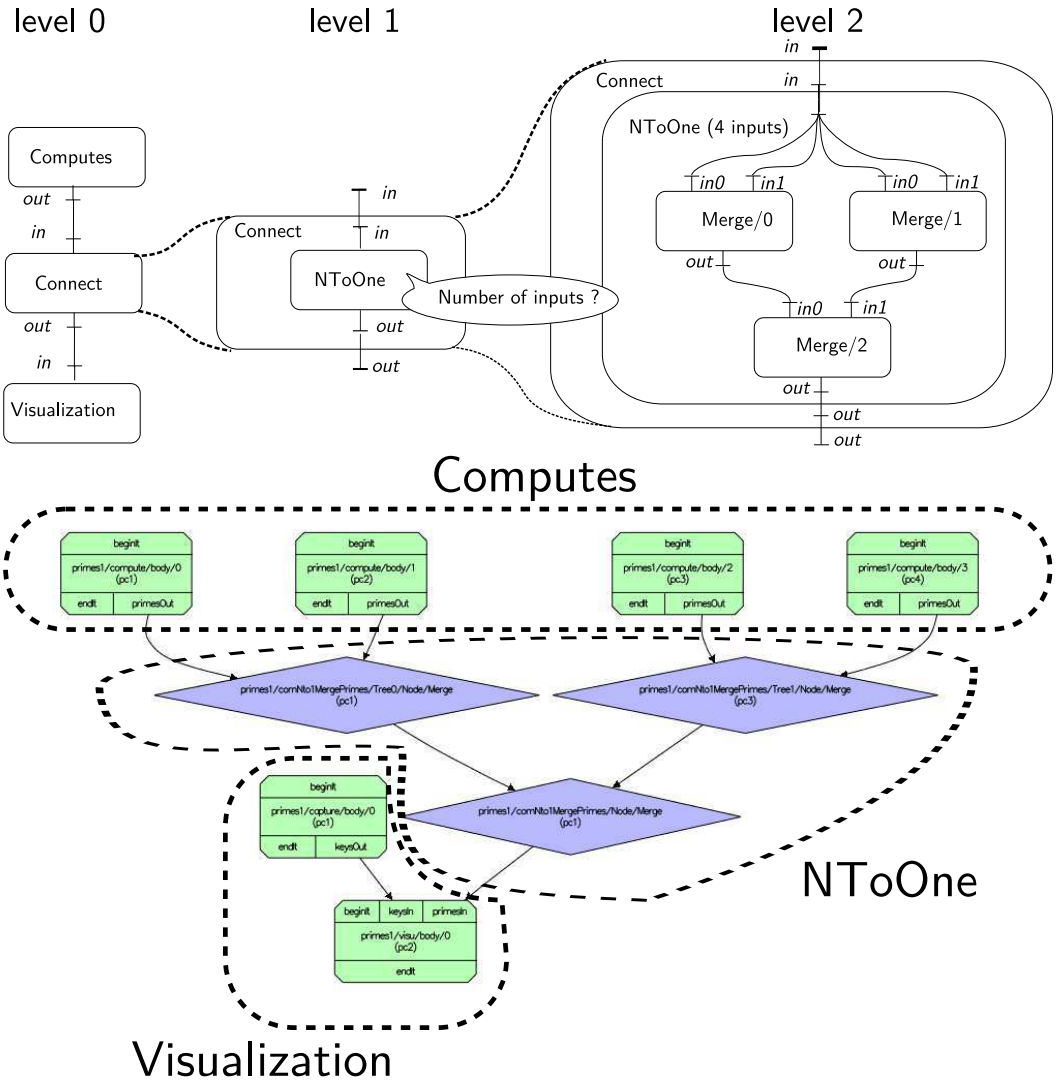


Figure 5: *[two columns width]* A component hierarchy (top) and the associated data-flow graph (bottom). Dashed sets show the composite components the graph elements are related to.

spawns only one module *Compute/0*, *Connect* will produce one point-to-point connection between *Compute/0* and *Render*.

For configuration controllers that depend on the state of other linked (directly or not) components, the traverse must obey a certain execution order to respect data dependencies. For instance, a component may define its number of children according to the number of modules of the metamodule it is linked to. It thus requires that the metamodule has defined its child modules before the controller can be called on the current component. This execution order could be given by the application developers. In this case, the developer has the responsibility to describe all dependencies between controllers to ensure the correct execution of the traverse. This option has two main drawbacks:

- Enumeration of all dependencies in case of thousands of components may be error-prone.
- Detection of dependencies implies a knowledge on each component implementation. This is not compatible with the hierarchical component concepts that push to hide implementations.

We use instead a simple algorithm that guarantees to complete the traverse when possible or return the list of misprogrammed components if some data dependencies cannot be solved whatever the execution order is. The traverse algorithm is a greedy process. The algorithm manages a queue of non-executed components, initialized with the top-level components of the application. For each component in this queue, the algorithm tries to execute the associated controller. If the controller is successfully executed, then all of its children are pushed in the queue. Otherwise, the algorithm restores the component initial state and push it at the end of the queue. The traverse ends successfully when the queue is empty. If no controller can be called on the remaining components in the list, then the algorithm stops in a fail state. The controller of the remaining components cannot be executed either because at least one of these components is misconfigured (a parameter is not instantiated for instance), or because a cycle of dependencies has been introduced when assembling the components.

The hierarchical component model only affects the front-end of FlowVR (Figure 4). The run-time environment is not modified. Components are written in C++ and compiled into shared libraries. An application is also a composite component compiled into a shared library. It can thus be reused in other applications without being recompiled. The FlowVR front-end loads the application and applies a sequence of traverses to produce the list of commands to start the modules and the instructions to forward to the different daemons to implement the data-flow graph.

4 Applications

We present in this section some non trivial VR applications we developed to highlight the benefits we experienced using FlowVR.

4.1 Grimage

The main application we developed is related to the Grimage platform[1] that associates a 16 projectors display wall, a PC cluster and a network of cameras. The goals is to develop VR applications using the camera network as the main input device. The application was developed over several years (2004-2008), involving various developers (about 10 different persons). We provide some snapshots of different versions of this application (Allard, Ménier, Boyer, & Raffin, 2005; Allard & Raffin, 2006; Allard, Ménier, Raffin, Boyer, & Faure, 2007) (Figure 7), but we strongly advice to refer to video materials[2] to get a quick and precise understanding of the application and evaluate the performance as well as the interaction level. The application is built from 3 main components (Figure 6) we present in the following sections.

4.2 Interactive 3D Modeling

The core element of this application is real-time 3D modeling used as an input device. 3D modeling computes a real-time model of the objects present into the acquisition space from a set of images shot from different videos cameras. We developed a parallel version of the Polyhedral Visual Hull algorithm (Franco & Boyer, 2008) that computes the complete

[1] http://grimage.inrialpes.fr

[2] http://flowvr.sourceforge.net/FlowVRGallery.html

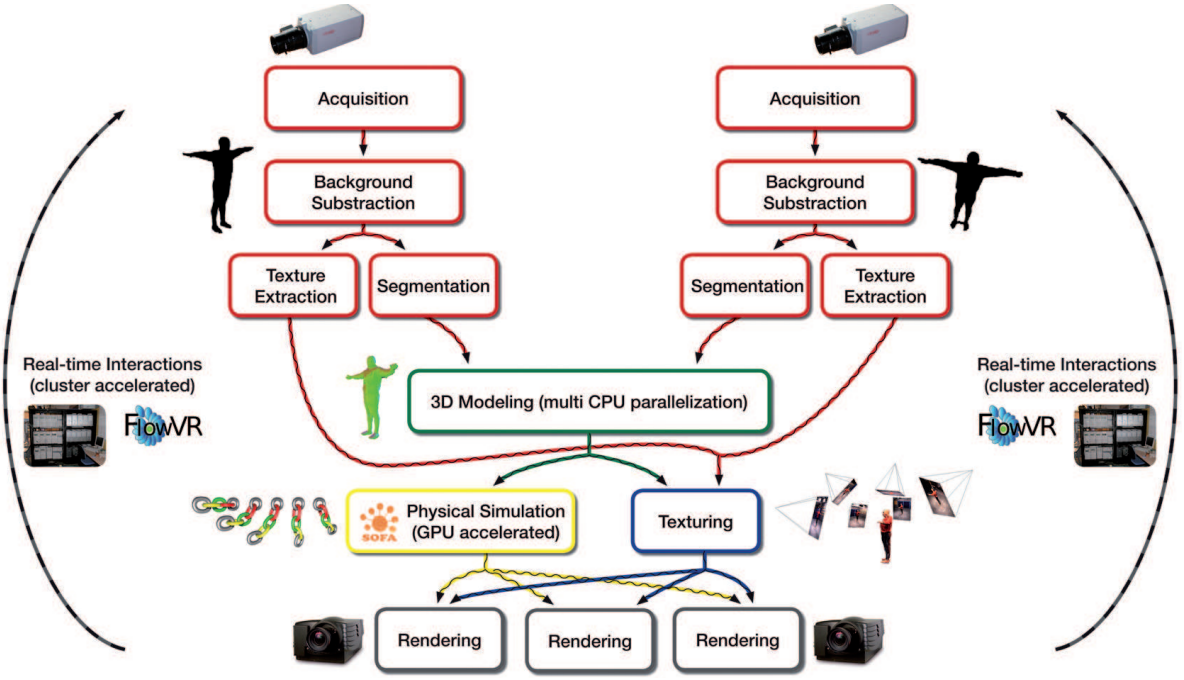


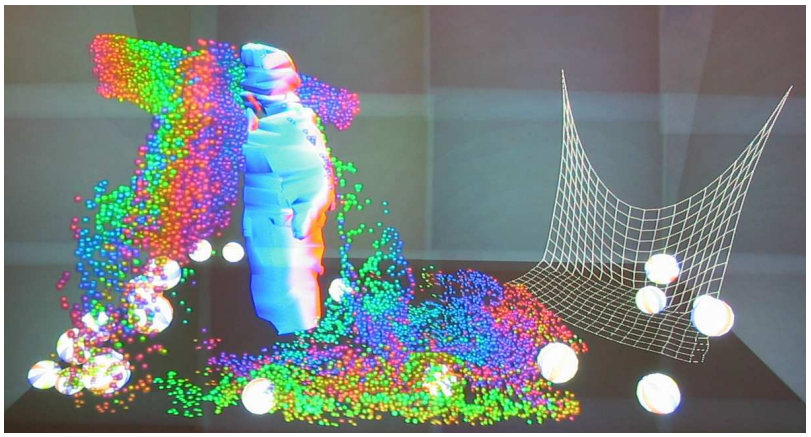
Figure 6: *[two columns width]* Overview of the application architecture as presented at Siggraph Emerging Technologies 2007.

and exact visual hull polyhedron with regard to silhouette inputs. The algorithm and its parallelization are presented in details in (Allard, Franco, Ménier, Boyer, & Raffin, 2006). The first parallelization attempt was based on MPI. But the application relies on an heterogeneous parallelism where tasks of different natures are assembled into a parallel pipeline. MPI led to a monolithic code using in excess MPI communicators to keep an acceptable level of modularity. The data-flow approach of FlowVR as well as the externalization of the network related issues and synchronisation policies proved more adapted for this heterogeneous parallelism. It became easier to detect erroneous programming choices, or to test different approaches playing with filters. This enabled to optimize the code more easily, leading to better performance than what was achieved with MPI. This code was initially developed in 2004 and after 4 years of improvements is still actively used. Amongst the most significant changes, models are now dynamically textured using the photometric data extracted from the images.

We also recently started to study collaborative interactions through 3D models by having two platforms (Grimage and its portable version) connected. The complete 3D modeling pipe-line is encapsulated in a high level component. Its specialization for a given target architecture is performed during the traverse process using a description of the target architecture file. For instance the mapping of the 2D image processing components on the cluster nodes (video acquisition, background subtraction, segmentation and texture extraction) is computed from the list of cameras to use. We added the second multi-camera platform by creating two instances of this 3D modeling pipe-line component associated with the correct configuration files.

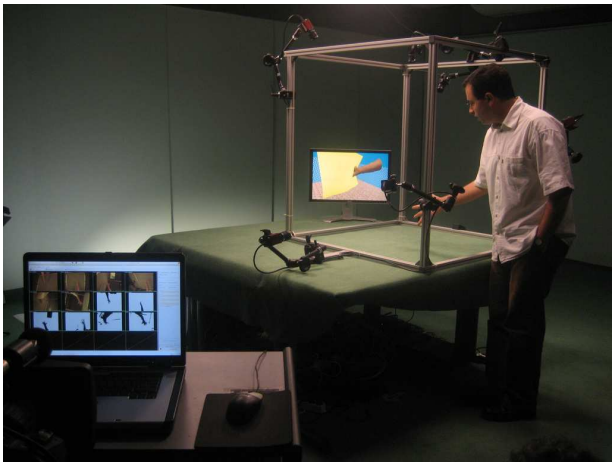
4.3 Distributed Physics Simulation

Combining the 3d modeling with a physics simulation is the next step to create a real-time full-body interaction system. As this requires many computations, we present here



(a)

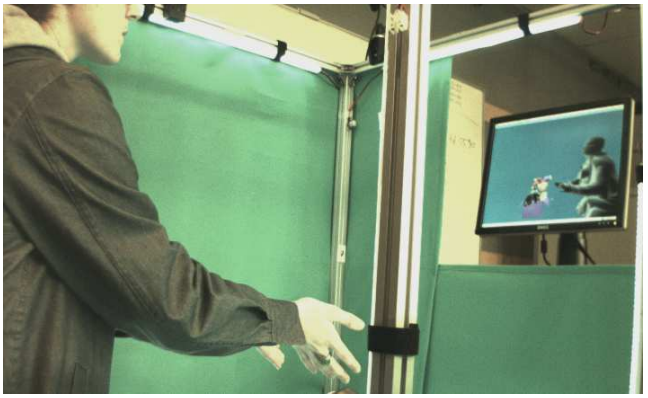
(b)



(c)

(d)

(e)



(f)

Figure 7: *[two columns width]* Various versions of 3D modeling based VR applications (2005-2008). (a) Distributed simulation, 3D model not textured (2005). (b) 3D model textured (2006). (c), (d) Siggraph 2007 Emerging Technologies demo with textured 3D models, shadows and reflections, and the SOFA simulation engine. (e), (f) Collaborative application using two 3D modeling platforms (2008).

a modular design of a physics simulation integrated within a FlowVR application.

The model relies on two main classes of modules, *animators* and *interactors* (Allard & Raffin, 2006). Animators own objects. They are responsible for updating their object states from the forces that apply to these objects. Objects are self-defined to ensure that adding or removing an object does not affect other objects. The forces applied to objects are computed by interactors, based on the object states they receive from animators. Each interactor is usually dedicated to one algorithm, for instance collisions between rigid objects. This enables a modular coupling of existing algorithms to build simulations combining the capabilities of each of these algorithms.

We designed an example simulation coupling a mass-spring based net simulation, a rigid bodies simulation, and a fluid simulation enabling two-way interactions between rigid objects and fluids (Carlson, Mucha, & Turk, 2004). Because the fluid simulation is computationally intensive, it was parallelized with MPI using block partitioning. Simulation and rendering reached 6.5 frames per second on a dual processor machine (1.6 GHz dual Opteron PC) with a scene consisting of 20 rigid objects, a $32 \times 64 \times 32$ fluid simulation, and a 20×20 mass-spring 2D net.

For the interactive execution, we added 3D modeling as an input device. A specific animator is dedicated to the 3D model. It is managed as a classical rigid object with one-way interactions (no force can affect it). The application was executed on the Grimage platform (Figure 7a). Eight hosts were dedicated to the parallel fluid simulation. Fluid rendering used particles. To avoid collapsing the network when sending the particles from each MPI process to the 16 rendering nodes (up to 200000 particles generated), the output filter of MPI process was set to dynamically route the grid cell content according to each projector's frustum. The application reached 18 fps, about three times faster than on a single machine, but with 16 times more pixels to compute and a dynamics 3D mesh to handle. The network is an important bottleneck. When all fluid particles are forwarded to all 16 rendering hosts, the frame rate is reduced to 1 fps. Dynamic particle routing enabled to significantly reduce this bottleneck to reach 18 fps.

This application, one of the largest we have developed so far, was built from a pool of 20 modules. The data-flow graph contained about 200 module instances, 4000 connections and 5000 filters.

4.4 FlowVR Render

A common feature in immersive VR applications is the use of several displays surfaces, covering a large fraction of the field of view or providing a high resolution. Supplying each of these displays with all the materials to be rendered is a challenging problem, particularly if the amount of information is large and updated frequently.

Distributed rendering combines two sub-problems: gathering data and sending it to the displays. The complexity of the gathering step depends on how much data is produced and whether it is distributed, while the send operation will have to scale depending on the number of displays. Designing a framework that is efficient in both cases, and particularly when combined, is an on-going research issue.

Gathering graphical data can be achieved at the pixel level. This sort-last approach (Molnar, Cox, Ellsworth, & Fuchs, 1994) is commonly adopted for rendering large data sets in scientific visualization. The data set is partitioned between different hosts. Each host locally renders an image, which are then composited in a reduction phase (Ma, Painter, Hansen, & Krogh, 1994) to obtain the final image.

Another approach consists in intercepting the calls to a graphics library, typically OpenGL, to encapsulate the data into messages that are then broadcasted to distant hosts for rendering. This approach was first developed for rendering unmodified OpenGL applications on a display-wall. The Chromium library (Humphreys et al., 2002) implements multiple optimizations to improve performance. For instance, it computes bounding boxes around objects to route them only towards the displays where they are to be rendered. But OpenGL makes it complex to merge data streams produced by multiple hosts. As OpenGL is based on a sequential state machine, commands must respect a strict ordering. Merging multiple streams together requires Chromium to track state changes and to use special commands defining the relative ordering of each stream.

Moving to higher level layers, applications can rely on a distributed scene-graph library (Roth, Voss, & Reiners, 2004). Thanks to high-level information such as change notifications and hierarchical groups, the distributed rendering implementation can then efficiently encode and transfer to the rendering the required updates.

On top of FlowVR we developed a distributed rendering library, called FlowVR Render. It adopts an intermediate strategy. Similarly to a scene graph, the application have to use a custom library to store and update the rendered information, however this description is at a lower level, and match the structures that are typically in use in advanced OpenGL codes (i.e. shaders, vertex buffers). The overall design is based on a protocol that makes it simple and efficient to transport and merge graphics primitives. In the rest of this section, we briefly describe this protocol, how it can be used to implement distributed rendering schemes, and finally provides a few examples of advanced applications where it makes a significant contribution.

4.4.1 Protocol

A scene is described as a set of independent primitives. Each contains the description of a batch of polygons and their rendering parameters. Thanks to the use of shaders, only a very small number of parameters are required, as the shaders control most of the visual appearance. Large resources such as textures or vertex attributes are often used by several primitives. Instead of duplicating these resources, they are encapsulated in a *shader*, *texture*, *vertex buffer* or *index buffer* identified by an unique ID. Each primitive then simply refers to the IDs of all the resources it uses. This level of description corresponds to the recommendation defined in the new OpenGL 3.0 standard, forcing the use of shader and buffer objects. However, there are a couple important differences.

Each primitive specifies its bounding box. This information is required to optimize communications using frustum culling, but is costly to recover from the other parameters (especially when using shaders that can change vertex positions).

In OpenGL, the primitives are necessarily rendered in the order that they are specified. FlowVR Render relaxes this ordering, allowing optimizations on the rendering side such as front-to-back sorting for fast Z-culling, or reducing texture and shader switches. An optional *order* parameter can however be specified for rendering algorithms requiring primitives to be processed in a given order.

This design leads to a very simple protocol, where only 3 types of operations (*add*, *remove*, *update*) can be applied on 2 types of elements (*primitive*, *resource*). This allows to efficiently process the data-streams to implement advanced routing or post-processing algorithms, as demonstrated below.

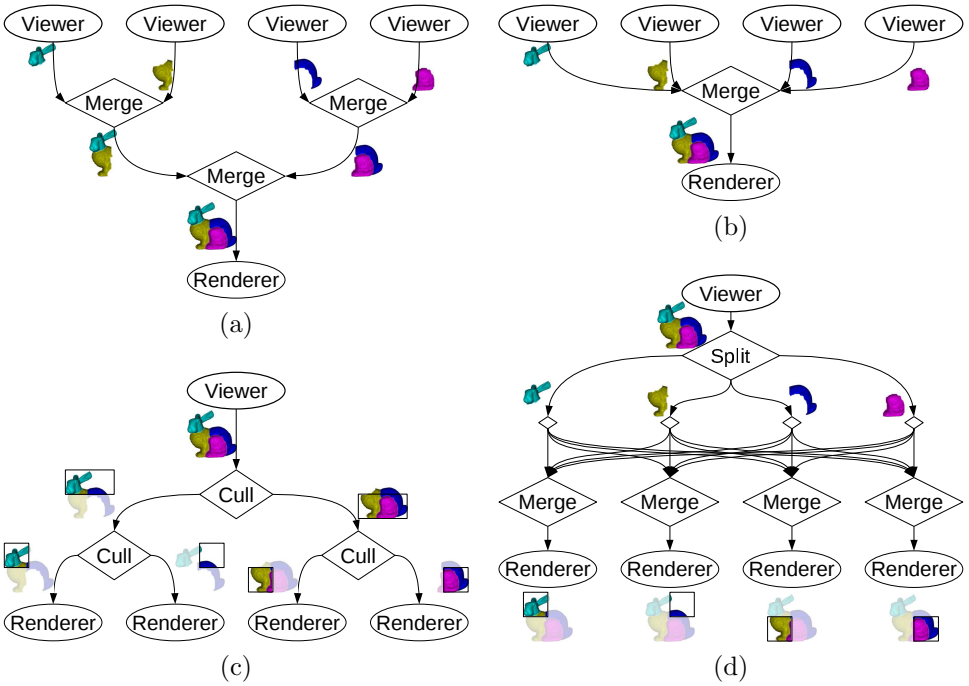


Figure 8: *[two columns width]* Different FlowVR Render rendering schemes

4.4.2 Distributed Rendering

We call *viewers* the tasks generating primitives and *renderers* the ones in charge of rendering them. Combining viewers, renderers and simple message processing tasks, FlowVR Render offers a large range of rendering schemes. The classical one is to have several viewers sending primitives to one renderer (Figure 8b). Primitives are merged into a single message handled to the render on the same hosts. It enables to keep the renderer code simple and ignorant of the number of viewers. Rather than to merge at a central point, it can easily be done progressively using a binary tree structure (Figure 8a). One viewer can also provide data to multiple renders using a simple broadcast, or, using again a binary tree communication scheme and culling filters to route each primitive where it is required based on its bounding box (Figure 8c). Performance tests (Allard & Raffin, 2005) show that FlowVR Render is significantly more scalable than Chromium regarding the number of viewers or renderers.

When large datasets must be continuously streamed to all renderers, such as the camera images used for texturing in the 3D modeling application, a more efficient design might be to rely on *peer-to-peer* exchange between renderers (Figure 8d). The viewer only sends each data once, but evenly split among renderers. These pieces will then be exchanged and merged back before rendering.

4.4.3 Usage Scenarios

One important limitation of FlowVR Render is the need to rewrite part of the rendering code of existing applications. However, in many cases the application rely on a broadly-used toolkit. Porting this toolkit to FlowVR Render requires a one-time effort that can thus enable the use of many applications. We validated this approach with VTK (Schroeder, Martin, & Lorensen, 2003) developing a VTK viewer. The low level OpenGL rendering layer of VTK was rewritten with FlowVR Render. It enables any VTK application to

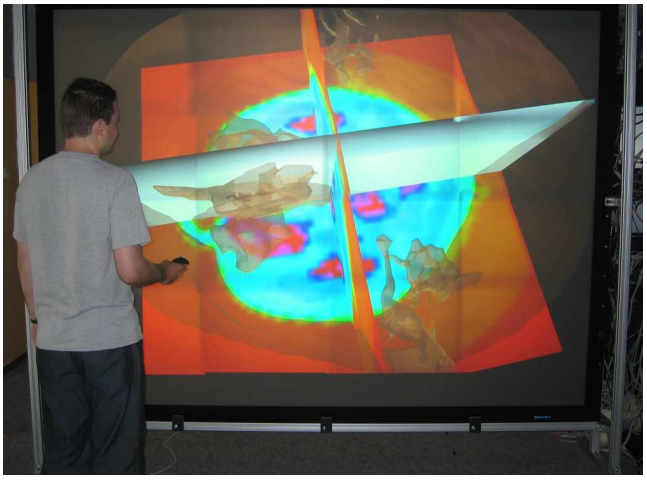


Figure 9: *[one column width]* Iso-surface extraction performed with VTK and multi-projector rendering with FlowVR Render.

render using FlowVR Render (Figure 9).

One important design feature of FlowVR Render is to separate the content of the objects present in virtual environment, specified by the viewers, from their composition and final rendering. In the previous application, this allowed for instance to transparently handle in the renderers each projector's calibrated transformation matrix and blending masks. But much more complex effects can be achieved by adding custom filters that will modify the data-streams based on the FlowVR Render protocol. As an example, we can implement a shadowing algorithm through a filter that duplicates all primitives so that they are first rendered onto a depth map, and change the shaders to use it when computing the lighting. To add a simple planar reflection effect, we can similarly duplicate each primitive while mirroring their transformation matrix. While these effects are relatively simple, the FlowVR Render design allows to apply then coherently in virtual environment containing objects from different viewers (scientific visualization toolkits, physics simulators, reconstructed real objects, etc.), each running at their own frequency and using different rendering abstractions. As demonstrated in Figure 7d, this can significantly improve the sense of coherency of the virtual environment, as relations between objects are more apparent through the shadows and reflections visible on each other. In traditional designs, this would have required to merge all the visualization code and datasets into a single program using an advanced rendering framework, which could prove very invasive and not always possible.

5 Discussion

We observed several limitations of FlowVR when developing large applications. We list in this discussion the more significant ones.

The benefits of the modularity provided by FlowVR are experienced for large applications. Using FlowVR for developing small prototypes is usually not efficient, except for already trained FlowVR developers. We also noticed that the hierarchical data-flow model requires a learning effort for developers that are used to the main loop model.

FlowVR can be verbose. It relies on C++ and adding a new module for instance

requires to develop the module run-time code as well as the component code. We are exploring various approaches to reduce this workload, using scripting languages on top of C++, or extracting the component code from a unified description of the run-time and component code.

The actual port concept is too limited. A port is only visible for the sibling or child components. When building a hierarchy, the user tends to reproduce most of the ports on the encapsulating components, so these ports stay available to connect to another component. We thus have a contradictory situations where the component hierarchy hides details, while the user tends to keep many ports exposed by reproducing them on ascendant components.

Some of the the composite patterns we provide can encapsulate a complex behavior. Similarly to design patterns or the STL library, making a proper use of these components can be difficult if the user does not properly understand their semantics.

FlowVR API enables the developer to directly allocate, read and write data in the shared memory segment. This enables to minimize the number of data copies, a time consuming task. This is the responsibility of the user to develop its own specialized layer for handling the types of messages his application requires. Such library is usually a good way to embed the specification of data encoding. Component developers can thus share this library to reduce the issues of compatibility between components. We use this approach for the FlowVR-Render messages for instance.

Data sometimes need to be converted when exchanged between modules. This can be performed using an intermediate conversion filter, avoiding a direct modification of the modules. For dynamic type conversions it is possible to define a message meta-data (stamp) to encode a data type, used to call the correct conversion function.

The data-flow graph of a given FlowVR application is static. There is no possibility to dynamically load, connect and start a new module. This is a feature often requested but our approach is to confine dynamic behaviors where it is required not to lose the benefits of a static graph. The full graph being available before starting the application, it can be displayed for debugging, or it can be analyzed for processing optimizations or error detections during traverses. Having a static graph does not prevent dynamic behaviors. A module can dynamically spawn processes or threads (FlowVR is thread-safe), as long as it takes care of scheduling and data sharing. A module or filter can also implement a classical client or server, a solution adopted by FlowVR VRPN (Limet & Robert, 2008). We are also evaluating how to enable a user to pause the application, add or remove components and resume again the newly configured application.

Monitoring a parallel application is complex and time consuming task. There is no exception with FlowVR. We intended to provide more informative tools, such as run-time listings of all components attached to a given daemon with frequency, messages queue size data, or a user programmable trace capture system and a simple trace visualizer. Advanced tools are required to better and more easily monitor FlowVR applications.

Processors evolved from single core to multi-core designs. FlowVR can be used to deploy applications on such architectures. Though the FlowVR daemon is multi-threaded, further optimizations are required to ensure high performance executions on large multi-core machines.

6 Conclusion

We presented FlowVR, a hierarchical component oriented middleware based on a data-flow paradigm. The goal was to support coupling various heterogeneous codes, possibly parallel, to build large interactive applications requiring multiple input, output and computing units. Experiences with large applications show that FlowVR meets this goal. It enforces a modular programming that reduces software engineering issues while enabling high performance executions on parallel architectures. It assists users by offering an environment separating module development, assembly, parametrization and instantiation on a given target architecture. Besides coupling, FlowVR also proved relevant to develop native parallel codes, like the FlowVR Render protocol for distributed rendering, a parallel physics simulation or the EPVH parallel 3D modeling algorithm.

FlowVR is distributed with several high level components encapsulating patterns that proved useful for several applications. The goal is to develop relevant patterns, generic yet usable, while avoiding their proliferation. An extra effort is still required to identify more advanced and more specialized design patterns. On-going work also focuses on providing the necessary abstractions, like a more flexible port system, to further improve modularity and help code developments.

Today FlowVR is distributed with support for distributed rendering through FlowVR-Render, VTK and VRPN. But our primary goal is to focus on the core FlowVR design and development. We expect external contributors to help us providing generic encapsulations of other existing tools.

References

Abram, G., & Treinish, L. (1995). An Extended Data-Flow Architecture for Data Analysis and Visualization. In *6th IEEE Visualization conference (VIS'95)* (pp. 263–270).

Ahrens, J., Law, C., Schroeder, W., Martin, K., & Papka, M. (2000). *A Parallel Approach for Efficiently Visualizing Extremely Large, Time-Varying Datasets* (Tech. Rep. No. LAUR-001630). Los Alamos National Laboratory.

Aldinucci, M., Coppola, M., Danelutto, M., Vanneschi, M., & Zoccolo, C. (2006). ASSIST as a Research Framework for High-performance Grid Programming Environments. In J. C. Cunha & O. F. Rana (Eds.), *Grid computing: Software environments and tools* (pp. 230–256). Springer.

Allard, J., Franco, J.-S., Ménier, C., Boyer, E., & Raffin, B. (2006). The GrImage Platform: A Mixed Reality Environment for Interactions. In *Fourth IEEE International Conference on Computer Vision Systems (ICVS'06)* (pp. 46–52).

Allard, J., Gouranton, V., Lecointre, L., Limet, S., Melin, E., Raffin, B., et al. (2004). FlowVR: a Middleware for Large Scale Virtual Reality Applications. In *10th International EuroPar Conference (EuroPar'04)* (pp. 497–505).

Allard, J., Gouranton, V., Lecointre, L., Melin, E., & Raffin, B. (2002). Net Juggler: Running VR Juggler with Multiple Displays on a Commodity Component Cluster. In *IEEE virtual reality* (pp. 275–276).

Allard, J., Ménier, C., Boyer, E., & Raffin, B. (2005). Running Large VR Applications on a PC Cluster: the FlowVR Experience. In *IPT & EGVE workshop* (pp. 59–68).

Allard, J., Ménier, C., Raffin, B., Boyer, E., & Faure, F. (2007). Grimage: Markerless 3D Interactions. In *ACM SIGGRAPH 07 emerging technologies.*

Allard, J., & Raffin, B. (2005). A Shader-Based Parallel Rendering Framework. In *IEEE visualization* (pp. 127–134).

Allard, J., & Raffin, B. (2006). Distributed Physical Based Simulations for Large VR Applications. In *IEEE virtual reality conference* (pp. 215–222).

Armstrong, R., Gannon, D., Geist, A., Keahey, K., Kohn, S., McInnes, L., et al. (1999). Toward a Common Component Architecture for High-Performance Scientific Computing. In *Proceeding of the 8th IEEE international symposium on high performance distributed computation* (pp. 115–124).

Baude, F., Caromel, D., & Morel, M. (2003). From Distributed Objects to Hierarchical Grid Components. In *International symposium on distributed objects and applications (DOA)* (pp. 1226–1242). Springer-Verlag.

Beckman, P. H., Fasel, P. K., Humphrey, W. F., & Mniszewski, S. M. (1998). Efficient Coupling of Parallel Applications Using PAWS. In *7th IEEE international symposium on high performance distributed computation* (p. 215).

Bierbaum, A., Hartling, P., Morillo, P., & Cruz-Neira, C. (2005). Implementing Immersive Clustering with VR Juggler. In *Computer graphics and geometric modeling workshop (TSCG'05)* (Vol. 3482/2005, pp. 1119–1128).

Bierbaum, A., Just, C., Hartling, P., Meinert, K., Baker, A., & Cruz-Neira, C. (2001). VR Juggler: A Virtual Platform for Virtual Reality Application Development. In *IEEE virtual reality* (p. 89).

Brodlie, K. W., Duce, D. A., Gallop, J. R., Walton, J. P. R. B., & Wood, J. D. (2004). Distributed and collaborative visualization. *Computer Graphics Forum*, *23*(2), 223–251.

Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., & Stefani, J.-B. (2006). The FRACTAL Component Model and its Support in Java: Experiences with Auto-Adaptive and Reconfigurable Systems. *Software Practice & Experience*, *36*(11-12), 1257–1284.

Carlson, M., Mucha, P. J., & Turk, G. (2004). Rigid fluid: Animating the interplay between rigid bodies and fluid. *ACM Transactions on Graphics*, *23*(3), 377–384.

Childs, H., Brugger, E., Bonnell, K., Meredith, J., Miller, M., Whitlock, B., et al. (2005). A contract based system for large data visualization. In *IEEE visualization* (pp. 191–198).

Cole, M. (1989). *Algorithmic skeletons: Structured management of parallel computation.* MIT Press.

Denis, A., Pérez, C., & Priol, T. (2003). Achieving Portable and Efficient Parallel CORBA Objects. *Concurrency and Computation: Practice and Experience*, *15*(10), 891–909.

Figueroa, P., Bischof, W. F., Boulanger, P., Hoover, H. J., & Taylor, R. (2008). InTml: A dataflow oriented development system for virtual reality applications. *Presence*, *17*(5), 492–511.

Foulser, D. (1995). IRIS Explorer: a Framework for Investigation. *Journal of ACM SIGGRAPH 95*, *29*(2), 13–16.

Franco, J., & Boyer, E. (2008). Efficient Polyhedral Modeling from Silhouettes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *31*(3), 414–427.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman Publishing Co., Inc.

Humphreys, G., Houston, M., Ng, R., Ahern, S., Frank, R., Kirchner, P., et al. (2002). Chromium: A Stream Processing Framework for Interactive Graphics on Clusters

of Workstations. In *Proceedings of ACM SIGGRAPH 02* (p. 693-702).

Jubertie, S., Melin, E., Vautard, J., & Lallouet, A. (2008). Mapping heterogeneous distributed applications on clusters. In *Euro-par 2008* (pp. 192–201).

Keahey, K., & Gannon, D. (1997). PARDIS: A Parallel Approach to CORBA. In *International symposium on high performance distributed computing* (pp. 31–39).

Kuck, R., Wind, J., Riege, K., & Bogen, M. (2008). Improving the AVANGO VR/AR Framework. In *Workshop Virtuelle und Erweiterte Realität.*

Lesage, J.-D., & Raffin, B. (2008). A Hierarchical Component Model for Large Parallel Interactive Applications. *Journal of Supercomputing*.

Limet, S., & Robert, S. (2008). FlowVR-VRPN: First Experiments of a VRPN/FlowVR Coupling. In *ACM Symposium on Virtual Reality Software and Technology (VRST'08)* (pp. 251–252).

Lohse, M., Winter, F., Repplinger, M., & Slusallek, P. (2008). Network-Integrated Multimedia Middleware (NMM). In *Proceedings of ACM Multimedia 2008 (ACM MM 2008).*

Ma, K.-L., Painter, J. S., Hansen, C. D., & Krogh, M. F. (1994). Parallel volume rendering using binary-swap compositing. *IEEE Computer Graphics and Applications*, *14*, 59–68.

Margery, D., Arnaldi, B., Chauffaut, A., Donikian, S., & Duval, T. (2002). OpenMASK: Multi-threaded or modular animation and simulation kernel or kit : a general introduction. In *VRIC* (pp. 101–110).

Mattson, T. G., Sanders, B. A., & Massingill, B. L. (2004). *A pattern language for parallel programming.* Addison Wesley.

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

Richart, N., Esnard, A., & Coulaud, O. (2007). Toward a Computational Steering Environment for Legacy Coupled Simulations. In *6th International Symposium on Parallel and Distributed Computing (ISPDC 2007)* (pp. 319–326). IEEE Press.

Roth, M., Voss, G., & Reiners, D. (2004). Multi-Threading and Clustering for Scene Graph Systems. *Computers & Graphics*, *28*(1), 63–66.

Schroeder, W., Martin, K., & Lorensen, B. (2003). *The Visualization Toolkit an object-oriented approach to 3D graphics, 3rd edition.* Kitware, Inc.

Serot, J., & Ginhac, D. (2002). Skeletons for Parallel Image Processing: an Overview of the SKIPPER Project. *Parallel Computing*, *28*(12), 1685–1708.

Soares, L. P., Raffin, B., & Jorge, J. A. (2008). PC Clusters for Virtual Reality. *The International Journal of Virtual Reality*, *7*(1), 67–80.

Wetzstein, G., Göllner, M., S.Beck, Weiszig, F., Derkau, S., Springer, J. P., et al. (2007). HECTOR - Scripting-Based VR System Design. In *Proceedings of ACM SIGGRAPH 07* (p. 143). ACM.

Wierse, A., Lang, U., & Rühle, R. (1993). Architectures of Distributed Visualization Systems and their Enhancements. In *Eurographics workshop on visualization in scientific computing.*

Zhang, K., Damevski, K., Venkatachalapathy, V., & Parker, S. G. (2004). SCIRun2: A CCA Framework for High Performance Computing. In *9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'04)* (pp. 72–79).

Zhang, L., & Parashar, M. (2006). Enabling Efficient and Flexible Coupling of Parallel Scientific Applications. In *Parallel and distributed processing symposium*

(IPDPS'06). (p. 10).