IPT & EGVE Workshop (2005)
R. Blach, E. Kjems (Editors)

Running Large VR Applications on a PC Cluster: the FlowVR Experience

Jérémie Allard Clément Ménier Edmond Boyer Bruno Raffin

Laboratoire Gravir, Laboratoire ID
CNRS/INPG/INRIA/UJF
INRIA Rhône-Alpes
655 avenue de l'Europe, 38334 Saint Ismier, France

Abstract

In this paper, we present how FlowVR enables the development of modular and high performance VR applications running on a PC cluster. FlowVR is a middleware we specifically developed targeting distributed interactive applications. The goal of the FlowVR design 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. FlowVR relies on an extended data flow model that enables to implement complex message handling functions like collective communications, or bounding box based routing. After a short presentation of FlowVR, we describe a representative application that takes benefit of FlowVR to reach a real time performance running on a PC Cluster.

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 [MF98, RVR04]. Several scientific visualization tools adopt a data-flow model [BDG*04]. An application corresponds to an oriented graph with tasks

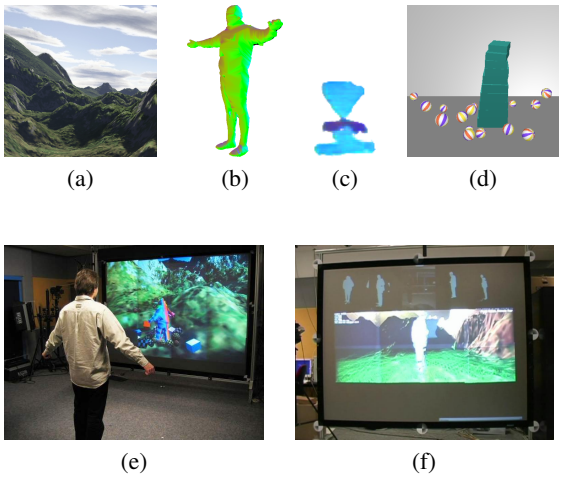


Figure 1: *Several applications developed with FlowVR such as a terrain rendering (a), a multi-camera 3D reconstruction (b), an object carving potter wheel (c) and a rigid body simulator (d), can be combined in a single large VR Application running on a cluster (e-f).*

at vertices and FIFO channels at edges. This graph clearly structures data dependencies between tasks. It eases task distribution on different processing hosts [AUR93]. To manage large distributed virtual worlds, networked virtual environments usually target only kinematic simulations of rigid objects [SZ00]. Each participant locally simulates the world for its zone of interest. The difficulty is then to ensure coherent interactions without slowing down the simulation due to strong synchronizations or a heavy network traffic. Approaches like dead-reckoning enable to extrapolate object position to absorb part of the network latency.

Hardware limitations have been tackled first by developing graphics supercomputers integrating dedicated hardware [MBDM97]. Focus was on increasing the rendering capabilities through different parallelization schemes [MCEF94]. Today, such approaches are facing difficulties to keep pace, regarding price and performance, with commodity component based platforms like graphics PC clusters [SFLS00]. But aggregating commodity components requires an extra effort on the software side. Chromium [HHN*02] 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 [CNSD*92, BJH*01, SG03]. 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 [GWN*03, MP04], 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 [AUR93], Net Juggler [AGMR02] or Avango [Tra99].

All the mentioned algorithms and tools are useful in different application scenarios. Large scale applications often requires 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 FlowVR [AGL*04], a middleware dedicated to distributed interactive applications. FlowVR reuses and extends the classical data-flow model. 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.

FlowVR comes with a complete tool suit to develop the modules and the applications, to map an application on a cluster, to launch it and control its execution. FlowVR also comes with tools for trace capture and visualization, to analyze an execution.

We have used FlowVR to develop several large VR applications taking advantage of up to 54 processors, with a display wall of 16 projectors and using multiple cameras as input devices. We present in this paper one representative application. We show the benefits of using FlowVR to favor the application modularity, to provide a model expressive enough to enable a large range of optimizations to reach a high performance. We also identify some basic patterns that have emerged from these developments and that proved well adapted for VR applications.

In section 2 we present FlowVR. Network patterns are presented in section 3. The applications and results are detailed in sections , 4, 5 and 6 before to conclude.

2. The FlowVR Middleware

2.1. Overview

FlowVR† is an open source middleware, currently ported on Linux and Mac OS X for the IA32, IA64, Opteron, and Power-PC platforms. In this section we present it main features. Refer to [AGL*04] for more details.

A FlowVR application is composed of two main parts, a set of modules and a data-flow network ensuring data exchange between modules. 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

† http://flowvr.sf.net

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.

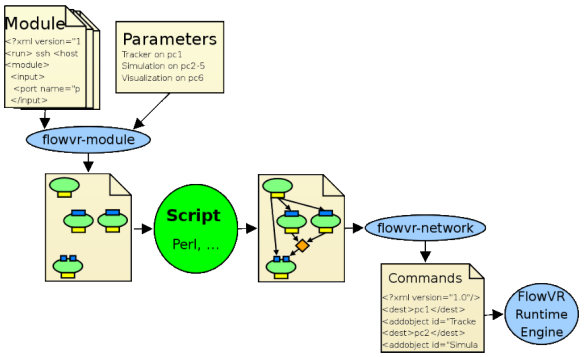


Figure 2: *The FlowVR application development chain. From modules (left) to the commands the controller forwards to daemons (right).*

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 2.5).
- 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.

Each implemented module is associated with an XML file that describes the module properties (Fig. 2). This file contains the path to the executable, the list of ports of the module and the command to launch it on a distant host. Templates and scripts can be used to ensure the genericity of this description. When designing an application, A second XML file is used to list the instances of modules involved in the application. For each module this list sets the module name, the host where it should be launched and the values of its different parameters. The *flowvr-module* utility parses these files to build (Fig. 2:

- the list of commands required to launch the modules,
- the list of all modules present in the application with their name, their list of ports and the host name they will run on. This list is the base for designing the FlowVR network.

Notice that FlowVR can handle commands that launch several modules at once. This is useful to include a paral-

lel code into a FlowVR application by having each process acting as a module.

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.

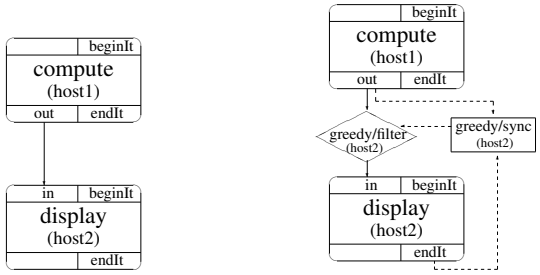
As such, a filter is more difficult to program than a module regarding message handling. Usually, a user only selects the filters it needs amongst the ones that come with FlowVR.

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.

To design a FlowVR network, the user writes a Perl script (Fig. 2). Using a procedural language enables a high level and compact network description. Numerous patterns that proved useful have been encapsulated into functions. If required, a user can complement the set of existing functions. This script takes as input the list of modules of the application and generates the list of FlowVR commands required to construct the network (two steps process involving the *flowvr-network* tool - Fig. 2).

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 the FlowVR network commands generated by the Perl script



(a) FIFO network. (b) Greedy network.

Figure 3: *Two different FlowVR networks to connect the compute and display modules. Full messages are carried over plain line connections, while stamps only are sent over dashed line connections.*

to the daemons that execute these commands to configure themselves (load plugins, set parameters, etc.). The execution of the application can then start.

2.5. Simple Example

Let us consider a simple example based on two modules called *compute* and *display*. Each one has a single port called *in* and *out* respectively. A first very simple application consists in running each module on a different host (*host1* and *host2*) and having a FIFO connection that enables *compute* to send each message it produces to *display* (Fig. 3(a)). The Perl script required to design this network is very simple:

```
use FlowVR::XML ':all';
parseInput();
addConnection('compute','out','display','in');
printResult();
```

The arguments to the `addConnection` method are the name and port of the source and destination modules. As this connection is a FIFO channel it forces the modules to run at the same frequency. This synchronous coupling scheme either reduces the framerate of the *display* module (if the *compute* module is the bottleneck), or introduces latency due to message bufferization.

To correct this behavior, VR applications often use a *greedy* pattern where the consumer uses the most recent data produced, all older data being discarded. This enables for instance to retrieve the last data produced from a tracker independently on the refresh rates of the producer and the consumer. FlowVR enables to implement such pattern without having to recompile the module. For that purpose we use a classical pattern based on a filter and a synchronizer (Fig. 3(b)). Each time the synchronizer *greedy/sync* receives an *endIt* message from *display*, it selects in its incoming buffer the newest stamp available and sends it to the filter *greedy/filter*. This filter waits to receive the message associated with that stamp, and forwards it to the *display* module. All older messages are discarded. This network is simply built replacing the *addConnection* call in the Perl script by

```
addGreedy('compute','out','display','in',
  getHosts('display'),getHosts('display'),
  'display','greedy');
```

In addition to the source and destination, we need to specify the location of the synchronizer and the filter (second line), as well as the module to get the *endIt* signals from and the prefix to use to name the created components. In this example we choose to map the filter and synchronizer on the *host2* of the *display* module. It favors system reactivity as requesting a new input value is only based on a local decision. Other configurations can be used. For instance mapping the filter and synchronizer on *host1* would save network bandwidth by avoiding messages that will be discarded to be sent over the network.

3. Network Patterns

In this section we present basic network patterns that proved useful for most of the VR applications we developed. Perl functions corresponding to these patterns are directly available.

3.1. Broadcast

Broadcasting a data is a classical collective communication pattern. It is often used in VR to ensure data retrieved from a driver are broadcasted to the multiple copies of the application, each one driving a different projector. The data received should be the same to ensure the coherency between the images computed by each copy.

We implemented a broadcast pattern based on a binary tree of routing nodes, each routing node having one input and two outputs. This approach has two advantages. First, only the size of the tree and not the filter have to be changed to adapt to the number of target modules. This is simply done by adjusting the parameters of the Perl function that sets such a tree. Second, the filters can be mapped on different processors to parallelize the execution of the broadcast.

3.2. Coherent Greedy Broadcast

The greedy behavior implemented for a simple one-to-one connection in section 2.5 can be generalized to the broadcast. But setting a greedy filter and synchronizer at the root of the broadcast tree impair reactivity as the data has to be broadcasted after being selected by the synchronizer, which increases the latency. To avoid this, we use a filter located at each leaf of the tree and a single synchronizer (Fig. 4). The synchronizer waits for all target modules to request a new data. It then sends the selection order to each filter that locally selects the right message to forward.

3.3. Merge

Gathering several messages into one is useful especially for computations distributed amongst several modules, each

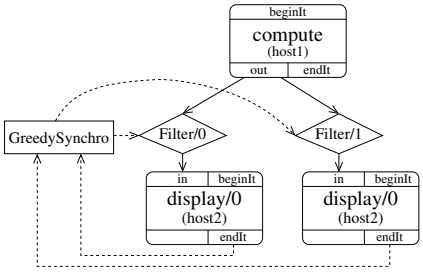


Figure 4: *Greedy broadcast using a centralized synchronizer to drive multiple filters.*

module computing only one part of the result. Similarly to the broadcast, we use an upside-down binary tree of *merge* filters.

The *merge* filter has two input ports and one output. It combines two incoming messages into a single one that is forwarded on the output. The default *merge* filter simply appends the different results. A modified filter can be used for more sophisticated combinations.

3.4. Stop/Start Control

Let us first consider a simple example where we want to allow an operator to activate or disactivate part of an application while running. Each module could be modified to retrieve this control information and handle it. This may however be difficult when dealing with modules written by others. We can take advantage in FlowVR of the fact that a module begins an iteration only when it has received a message on each of its connected input port. Blocking an input flow implies stopping the computation of the module. We developed a controlling filter to implement this blocking mechanism. Such filter let flows pass if the value on its control port is true. This filter simply discards any messages when the value on its control port is false. This allows for messages not to accumulate and for restarting the program with the most recent data. We usually associate such filters with a module that displays GUI elements such as toggle buttons. Clicking the button sends a signal to the corresponding filter.

3.5. Frequency Control

By default modules usually does not control their iteration rate. However it can be useful for avoiding useless iterations that consume processor time and network bandwidth if they are not discarded before being send by a greedy filter. To enable controlling a module frequency without having to modify the code of the module itself we use a *max frequency* synchronizer (Fig. 5). This synchronizer gets the frequency value on one input port. Each time it receives a request for a new iteration from the module, it may delay its answer (sent on the *beginIt* port of the module) to enforce that the target frequency is respected.

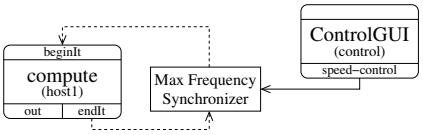


Figure 5: *Frequency control pattern using the MaxFrequency synchronizer.*

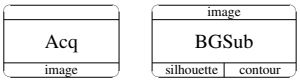


Figure 6: *(a) Acquisition module retrieving color images from a camera. (b) Silhouette extraction module in charge of computing the silhouette and its vectorized contour from a color image.*

4. Application Components: VR Modules

In this section, we report on the FlowVR use for developing a large VR application. After briefly presenting the application, we describe the different modules developed following the FlowVR approach.

4.1. Application overview

To demonstrate how FlowVR helps in developing large VR applications we chose our most representative application. It consists in implementing several user interactions with visualization on a display wall. It is made of three main parts: 3D input device, advanced interactions and distributed rendering. Each part is relatively complex and computationally demanding. Basically, it extends the markerless 3D modeling described in [ABF*04]. The input device retrieves 3D user shape information through state-of-the-art vision technics, integrating cues from multiple cameras. This information is then used for implementing user interactions. We focus on two advanced interactions: virtual carving and collision with simulated rigid bodies. Finally visualization is done on a 4×4 multi-projectors display wall. Note that we do not immerse the user in the environment but instead display his 3D image in the virtual world.

4.2. Input/Acquisition Modules

Retrieving 3D information from the user is done through vision-based methods using multiple cameras. It is a computationally intensive process that requires several processing steps. We describe in this section how each step is implemented as a module. 3D modeling algorithms not being in the scope of this paper, we do not detail the module implementations. For more details, refer to [ABF*04].

Images are first acquired on each PC driving a camera by the **Acq** module (Fig. 6(a)). It is implemented as a FlowVR loop retrieving, at each iteration, a new image from the camera, transforming it into a message then sending the message on its **image** port. Color images will then need to be processed to extract the silhouette, i.e. the image region corresponding to the user. This process is implemented in a **BGSub** module (Fig. 6(b)). For each frame, it reads on its **image** port a color image and applies a background subtraction algorithm [HHD99] on it. It results in the silhouette, a black and white image sent on its **silhouette** port. This module then vectorizes the contour of the silhouette and sends the resulting message on its **contour** port.

Note here we use two different modules, **Acq** and **BGSub**, where we could have used only one. Separating those modules presents two main advantages. The first one concerns reusability. The **Acq** module can be used in other applications not requiring silhouettes. Second, both modules can be mapped on two different hosts if the machine in charge of the camera acquisition is computationally limited. Reversely, the overhead of running two modules on the same host rather than a single one doing all the work, is limited as a FlowVR message exchange is a simple pointer exchange.

As 3D user shape will be used in different contexts – visualization and interaction – we use two vision technics, each computing a different information with specific interests. The first one, well suited for visualization, consists in computing an approximation of the user surface (mesh). We implemented the distributed algorithm presented in [ABF*04] into a parallelized module, **Rec** (Fig. 7(a)). It takes vectorized silhouettes as input and outputs partial meshes. Another method, better suited for interaction, consists in extracting volumetric information by discretizing space in voxels. The **Voxel** module (Fig. 7(b)) reads at each frame the new silhouettes and computes the occupancy grid. As some other modules require distance-to-shape information, we implemented the Euclidean Distance Transform [ST94] in the module **EDT** (Fig. 7(c)). It computes for each cell of the grid the distance vector to the user surface.

Figure 7: *(a) Reconstruction module implementing [ABF*04]. This module is distributed on several hosts, each one computing its part of the mesh from the silhouette contour of each camera. (b) Volumetric reconstruction using a voxel grid. (c) Euclidean Distance Transform computing the distance grid from the voxel grid.*

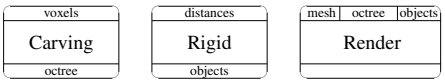


Figure 8: *(a) Carving module implementing a virtual pottery interaction. Voxels are used to detect which portion of space should be altered (carved or painted for example). (b) Rigid body simulator handling collision with the user thanks to the distance grid. (c) Visualization module rendering the virtual environment containing the terrain and the different objects: user mesh, pottery octree and rigid objects.*

4.3. Interaction and Visualization Modules

Using the acquired 3D information on the user, we can implement user interactions. To this aim we developed two interactive modules.

The first one, **Carving** (Fig. 8(a)), consists in a virtual pottery application where the user carves/paints a virtual object (octree) by using its full body. The program retrieves for each iteration the voxel occupancy grid and uses it to update the octree according to the interaction mode (carving, adding or painting matter). At each iteration, it results in an octree, a color being attached to each cell.

The second module, **Rigid** (Fig. 8(b)), consists in a physically based rigid body simulator. We use a publicly available implementation of the algorithm from Guendelman *et al.* [GBF03]. The program handles a set of rigid objects and simulates their dynamics. To port this application into a FlowVR module, we build at each iteration a message containing the new positions and shapes of the simulated objects in the virtual world. This message is sent on the **objects** output port. Support for external objects is done by reading their distance grid information and adding those objects in the simulation.

Finally the virtual environment is displayed through a **Render** module based on VRJuggler 2[‡]. (Fig. 8(c)). It contains 3 input ports: **mesh** for displaying the user reconstructed surface, **octree** for visualizing the virtual sculpture and **objects** for visualizing the rigid objects. We added FlowVR functions to retrieve input data inside the VR Juggler application class methods. Note that this module will be replicated on several hosts for visualization on a display wall. Internal VR Juggler communications to ensure the swaplock is transparent to FlowVR.

5. Application Design

Having constructed our pool of modules, building the application consists in defining the FlowVR network. We describe how to couple modules in order to obtain a good trade-off between performance and coherency for the application.

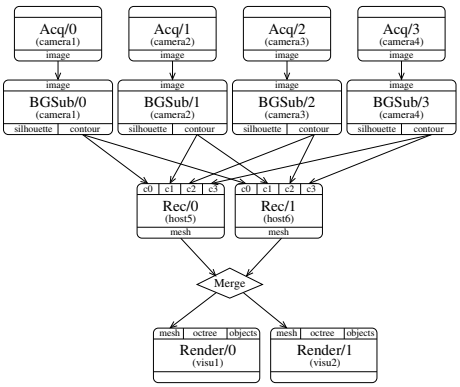


Figure 9: *FlowVR network graph for FIFO coupling of the mesh reconstruction with the distributed visualization.*

Let us start by coupling the mesh reconstruction with the visualization in order to display the 3D user shape in the virtual world. Figure 9 presents the graph when using simple connections between modules with 4 cameras and 2 projectors. Note that modules are instantiated several times (4 **Acq**, 4 **BGSub** and 2 **Render**). This FIFO coupling raises a frequent issue: visualization is constrained to run at the reconstruction speed, around 30 frames per second, as it waits for a new model at each iteration. This leads to jitters especially when the user moves the rendering point of view. To cope with this issue, we replace the FIFO strategy by a coherent greedy broadcast strategy for visualization input data.

Figure 10 presents the network graph for the full application with the two interaction modules. It uses a greedy broadcast pattern for each visualized data. The FIFO coupling raises another common issue: the simulations (rigid in this case) run at the same frequency as the input data (cameras). Here again, using a greedy filtering between the input devices and the simulation allows the simulation to run at a higher pace. This leads to more precise and more stable simulations, as they are allowed to use smaller timesteps.

6. Results

The full application was tested in our experimental setup, the Grimage platform[§]. It consists in 6 cameras and 4×4 projectors display wall. Those devices are driven by 2 interconnected clusters: 11 dual-Xeon at 2.6 GHz for the vision part of the application and 16 dual-Opteron at 2 GHz for the visualization part. Note that FlowVR supports heterogeneous hardware in the same application.

The complete graph of the application used during the

[‡] http://www.vrjuggler.org

[§] http://www.inrialpes.fr/grimage/

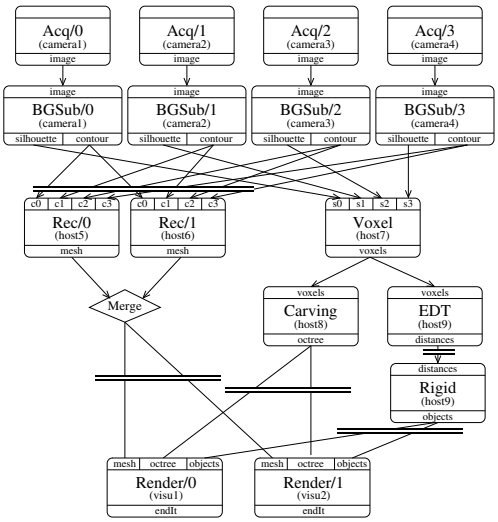


Figure 10: *FlowVR network graph for the full application graph. Simulations and input device are coupled by FIFO connections. Coherent broadcast greedy patterns (Fig. 4), represented with double horizontal bars, are used on each visualized data as well as on Rec and Rigid inputs.*

tests is presented in figure 12. It corresponds to the network graph shown in figure 10 with additional rendering modules for the 16 projectors as well as an additional *MainControl* GUI module to activate each parts of the application (through connection and control filters shown in red). Figure 11 shows the interactions in the running application. Videos are also available[¶].

Figure 13 presents the frequency of each part of the application during one execution of about 3 minutes where several people where entering and leaving the interaction space. Using the previously presented asynchronous coupling patterns, FlowVR was able to execute each component at maximum speed. While the reconstruction speed is fluctuating between 20 and 30 updates per second depending on the complexity of the reconstructed mesh, the interaction modules (Voxel, EDT, Rigid) are able to maintain a constant frequency, allowing for smooth interactions. After a short warm-up phase, The rendering part is able to refresh the display at least 75 times per second. When a part of the application is not useful the control mechanism can disactivate it, thus freeing shared resources such as network bandwidth to other tasks. This is used during this experiment when the collision-related modules where disabled for a short time.

Our one year experience with FlowVR has demonstrated its great usability and its efficiency for developing large and complex VR applications. On the Grimage platform the most

¶ http://www.inrialpes.fr/grimage/gallery.php

(a)

(b)

Figure 11: *(a) Carving interaction experiment, with the silhouette images displayed on top. (b) Rigid body simulation with collision with the user's body.*

complex application[¶] (not presented due to its complexity, but reusing most of the modules and patterns presented in this paper) was developed by 4 persons in about 6 months. It integrates around 50 modules from different persons and teams. Using this module pool we obtain more than 200 processes. Using about 2000 lines of Perl code we have generated a graph with more than 5000 objects (connections, filters and synchronizers). While large, the application is still performant as it efficiently uses the available computational ressources. This experience clearly demonstrates the efficiency of FlowVR for developing large, distributed VR applications.

7. Conclusion

We presented FlowVR and its advanced use for developing parallel VR applications. FlowVR is based on an extended

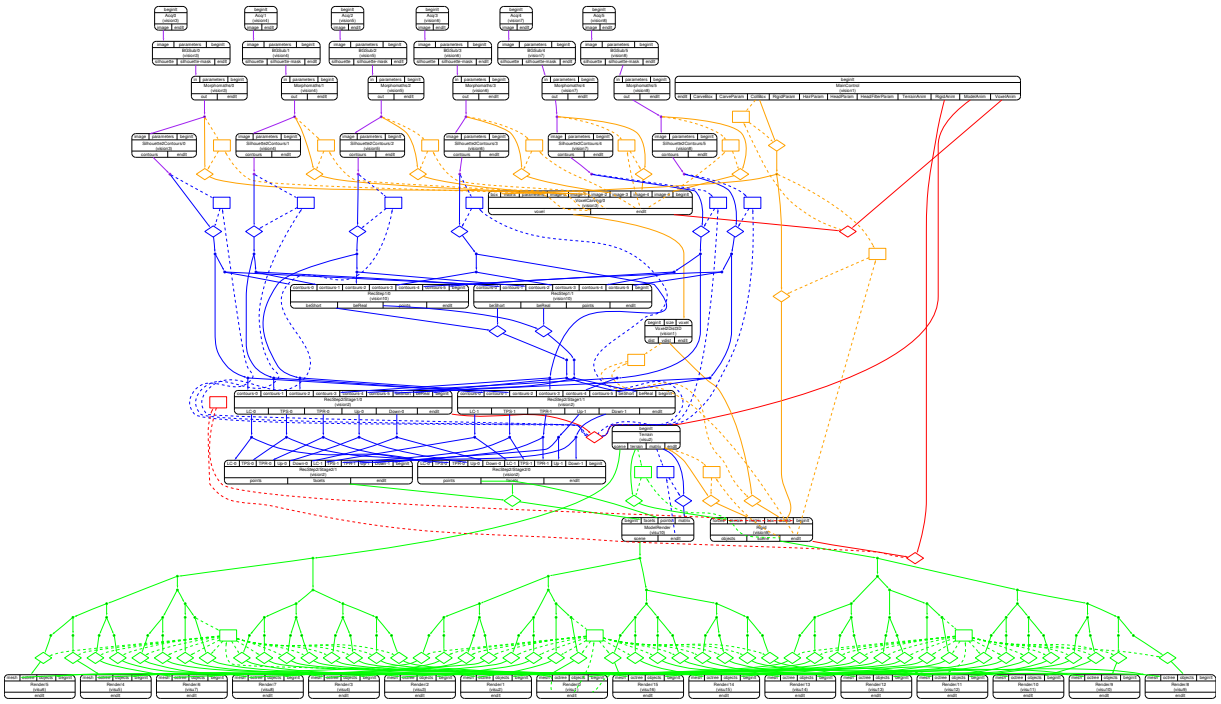


Figure 12: *Complete graph of the application with* 6 *cameras and* 16 *projectors.*

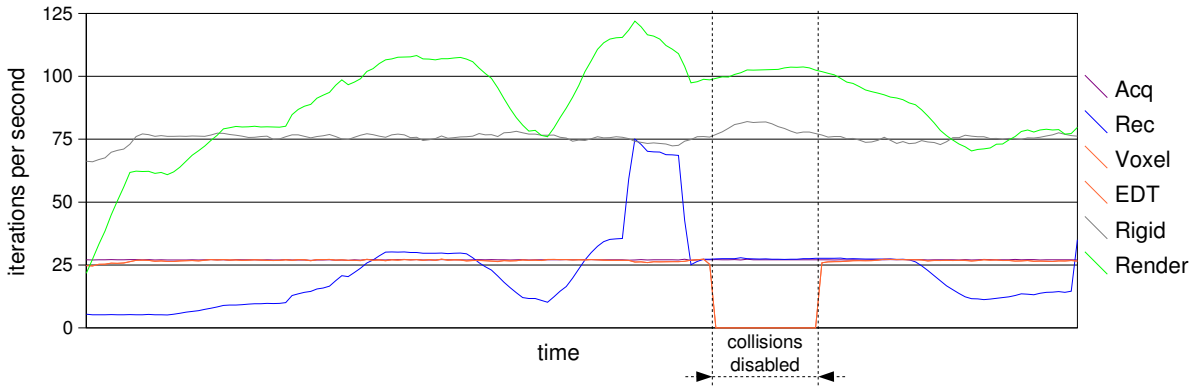


Figure 13: *Frequencies of each part of the application during one execution.*

data flow model. Developing an application is a two step process. Modules are first developed. Modularity is enforced by letting modules run in their native process or thread and keeping modules unaware of the surrounding network. The FlowVR network that connects modules is developed in a second step. Specific network components, like routing nodes, filters and synchronizers, can be used to implement complex message handling functions. In particular, filters enable to implement behaviors that cannot be programmed at the module level due to the voluntarily restricted module programming interface.

FlowVR was used to develop several applications using multiple cameras and multiple projectors as I/O devices. Interactive executions were achieved by distributing the acquisition, computation and rendering tasks on a cluster. These developments showed that FlowVR leads to modular and high performance applications. In particular, we were able to manage the complexity of the applications beside the fact that several people were involved in code development, and that some modules were encapsulating complex codes that we did not develop (VR Juggler, rigid body simulation, etc.). The development environment proved adapted to han-

dle large applications. In particular, it enables compact network descriptions and to manage application launching even if modules require very different launching commands.

Beside VR Juggler, we are also using a shader-based parallel rendering framework developed on top of FlowVR [AR05].

As the FlowVR use progresses, we expect to build public libraries of modules and filters. On going work focus on a library dedicated to image handling.

References

[ABF*04] ALLARD J., BOYER E., FRANCO J.-S., MÉNIER C., RAFFIN B.: Marker-less Real Time 3D Modeling for Virtual Reality. In *Proceedings of the Immersive Projection Technology Workshop* (Ames, Iowa, May 2004).

[AGL*04] ALLARD J., GOURANTON V., LECOINTRE L., LIMET S., MELIN E., RAFFIN B., ROBERT S.: FlowVR: a middleware for large scale virtual reality applications. In *Euro-Par 2004 Parallel Processing: 10th International Euro-Par Conference* (Pisa, Italia, August 2004), pp. 497–505.

[AGMR02] ALLARD J., GOURANTON V., MELIN E., RAFFIN B.: Parallelizing Pre-rendering Computations on a Net Juggler PC Cluster. In *Immersive Projection Technology Symposium* (Orlando, USA, March 2002).

[AR05] ALLARD J., RAFFIN B.: A shader-based parallel rendering framework. In *IEEE Visualization Conference* (Minneapolis, USA, October 2005).

[AUR93] A.WIERSE, U.LANG, RÜHLE R.: Architectures of Distributed Visualization Systems and their Enhancements. In *Eurographics Workshop on Visualization in Scientific Computing* (Abingdon, 1993).

[BDG*04] BRODLIE K. W., DUCE D. A., GALLOP J. R., WALTON J. P. R. B., WOOD J. D.: Distributed and collaborative visualization. *Computer Graphics Forum 23*, 2 (2004).

[BJH*01] BIERBAUM A., JUST C., HARTLING P., MEINERT K., BAKER A., CRUZ-NEIRA C.: VR Juggler: A Virtual Platform for Virtual Reality Application Development. In *IEEE VR 2001* (Yokohama, Japan, March 2001).

[CNSD*92] CRUZ-NEIRA C., SANDIN D. J., DEFANTI T. A., KENYON R. V., HART J. C.: The Cave Audio VIsual Experience Automatic Virtual Environement. *Communication of the ACM 35*, 6 (1992), 64–72.

[GBF03] GUENDELMAN E., BRIDSON R., FEDKIW R.: Nonconvex rigid bodies with stacking. *ACM Trans. Graph. (Proceedings of ACM SIGGRAPH 03) 22*, 3 (2003), 871–878.

[GWN*03] GROSS M., WUERMLIN S., NAEF M., LAMBORAY E., SPAGNO C., KUNZ A., KOLLER-MEIER E., SVOBODA T., GOOL L. V., S. LANG K. S., MOERE A. V., STAADT O.: Blue-C: A Spatially Immersive Display and 3D Video Portal for Telepresence. In *Proceedings of ACM SIGGRAPH 03* (San Diego, 2003).

[HHD99] HORPRASERT T., HARWOOD D., DAVIS L.: A Statistical Approach for Real-time Robust Background Subtraction and Shadow Detection . In *IEEE ICCV'99 FRAME-RATE WORKSHOP* (1999).

[HHN*02] HUMPHREYS G., HOUSTON M., NG R., AHERN S., FRANK R., KIRCHNER P., KLOSOWSKI J. T.: Chromium: A Stream Processing Framework for Interactive Graphics on Clusters of Workstations. In *Proceedings of ACM SIGGRAPH 02* (2002), pp. 693–702.

[MBDM97] MONTRYM J. S., BAUM D. R., DIGNAM D. L., MIGDAL C. J.: InfiniteReality : A Real-Time Graphics System. In *Proceedings of ACM SIGGRAPH 97* (Los Angeles, USA, August 1997), pp. 293–302.

[MCEF94] MOLNAR S., COX M., ELLSWORTH D., FUCHS H.: A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics and Applications 14*, 4 (July 1994), 23–32.

[MF98] MACINTYRE B., FEINER S.: A distributed 3D graphics library. In *Proceedings of ACM SIGGRAPH 98* (1998), Cohen M., (Ed.), Addison Wesley, pp. 361–370.

[MP04] MATUSIK W., PFISTER H.: 3D TV: A Scalable System for Real-Time Acquisition, Transmission, and Autostereoscopic Display of Dynamic Scenes. In *Proceedings of ACM SIGGRAPH 04* (2004).

[RVR04] ROTH M., VOSS G., REINERS D.: Multithreading and clustering for scene graph systems. *Computers & Graphics 28*, 1 (2004), 63–66.

[SFLS00] SAMANTA R., FUNKHOUSER T., LI K., SINGH J. P.: Hybrid Sort-First and Sort-Last Parallel Rendering with a Cluster of PCs. In *ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware* (August 2000).

[SG03] SCHAEFFER B., GOUDESEUNE C.: Syzygy: Native PC Cluster VR. In *IEEE VR Conference* (2003).

[ST94] SAITO T., TORIWAKI J.: New algorithms for euclidean distance transformations of an n-dimensional digitised picture with applications. *Pattern Recognition 27*, 11 (1994), 1551–1565.

[SZ00] SINGHAL S., ZYDA M.: *Networked Virtual Environments - Design and Implementation*. ACM SIGGRAPH Series. ACM Press Books, 2000.

[Tra99] TRAMBEREND H.: Avocado: A distributed virtual reality framework. In *Proceedings IEEE Virtual Reality 99 ConferencE* (March 1999), L. Rosenblum P. A., Teichmann D., (Eds.), pp. 14–21.