FlowVR: A Framework For Distributed Virtual Reality Applications

Thomas Arcila[*] Jérémie Allard[†] Clément Ménier[‡] Edmond Boyer[§] Bruno Raffin[¶]

Laboratoire ID & Laboratoire GRAVIR
Grenoble
CNRS/INPG/INRIA/UJF

Abstract

This paper introduces the FlowVR suite, a set of softwares targeted at virtual reality applications distributed on clusters or grid environments. The FlowVR middleware supports coupling of heterogeneous parallel codes and is component oriented to favor code reuse. After introducing the FlowVR main concepts, we details the different tools associated and several applications.

Keywords: Virtual reality, PC Cluster, Parallel Computing

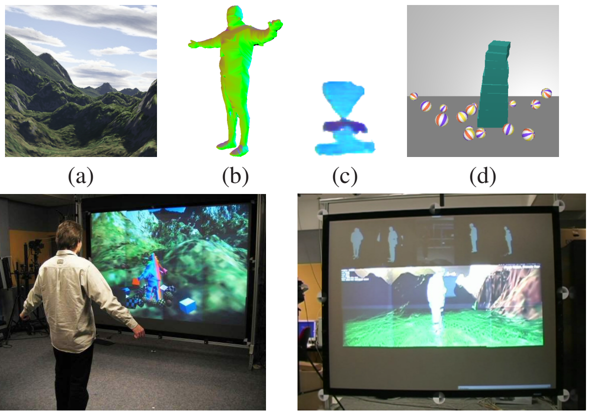


Figure 1: Several applications developed with FlowVR such as a terrain rendering (a), a multi-camera 3D reconstruction (b), an object carving potter wheel (c) and a rigid body simulator (d), can be combined in a single large VR Application running on a cluster (e-f).

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 [13, 18]. Several scientific visualization tools adopt a data-flow model [7]. An application corresponds to an oriented graph with tasks at vertices and FIFO channels at edges. This graph clearly structures data dependencies between tasks. It eases task distribution on different processing hosts [5]. To manage large distributed virtual worlds, networked virtual environments usually target only kinematic simulations of rigid objects [21]. 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.

Hardware limitations have been tackled first by developing graphics supercomputers integrating dedicated hardware [16]. Focus was on increasing the capabilities through different parallelization schemes [15]. Today, such approaches are facing difficulties to keep pace, regarding price and performance, with commodity component based platforms like graphics PC clusters [19]. But aggregating commodity components requires an extra effort on the software side. Chromium [11] 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 [8, 6, 20]. 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 [10, 14], 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 [5], OpenMask [1] or Avango [22].

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 the FlowVR suite [2, 4], a software suite dedicated to distributed interactive applications.

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

[*]e-mail: thomas.arcila@imag.fr
[†]e-mail: jeremie.allard@imag.fr
[‡]e-mail: clement.menier@inrialpes.fr
[§]e-mail: edmond.boyer@inrialpes.fr
[¶]e-mail:bruno.raffin@imag.fr

VTK FlowVR (section 4) that enables rendering VTK applications with FlowVR Render.

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

2 The FlowVR Middleware

2.1 Overview

FlowVR is an open source middleware, currently ported on Linux and Mac OS X for the IA32, IA64, Opteron, and Power-PC platforms. In this section we present its main features. Refer to [2] for more details.

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

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

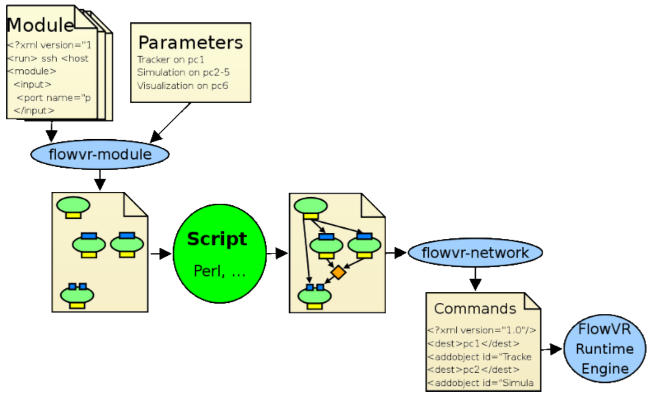
The FlowVR network defined between modules can implement simple module-to-module connections as well as complex message handling operations. For instance the network can implement synchronizations, data filtering operations, data sampling, dead reckoning, frustum culling, collective communications schemes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

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

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

2.2 Messages

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



```
<?xml version="1
<run> ssh <host
<module>
  <input>
   <port name="p
  </input>
```

```
<?xml version="1.0"/>
<dest>pc1</dest>
<addobject id="Tracke
<dest>pc2</dest>
<addobject id="Simula
```

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

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* (Fig. 2) utility parses these files to build:

- 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 parallel 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.

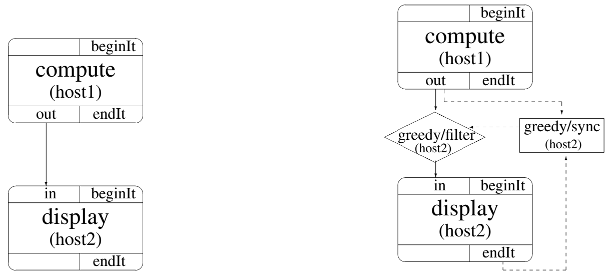
We also distinguish another special class of filters, called *synchronizers*. A synchronizer implements coupling policies by centralizing data from other filters or modules to take a decision that will then be executed by other filters. A synchronizer differs from standard filters because all input and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



(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.

```
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 FlowVR Render

FlowVR Render [4] is a shader based parallel rendering framework relying on FlowVR. It takes advantage of the power offered by graphics clusters to drive display walls or immersive multi-projector environments like Caves. It defines graphics primitives using shader programs to propose a high performance communication protocol:

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

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

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

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

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

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

4 VTK-FlowVR

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

5 Making Development Easier

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

- FlowVR-GLGraph, an OpenGL based FlowVR network viewer, gives users the opportunity to display the network of the instanciated application. It features color handling, zoom, hiding of uninteresting parts to make the network more readable (Fig. 4).

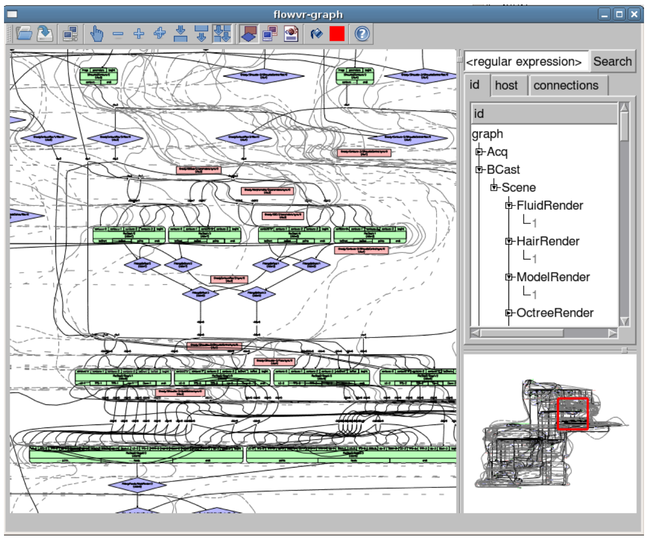


Figure 4: FlowVR OpenGL graph viewer

- FlowVR-GLTrace is a trace visualization tool. FlowVR supports capture of predefined or user-defined events. Capture is performed with minimal impact on the execution performance. Once stored on disk, FlowVR-GLTrace enable to process and display the execution trace. This tool helps debugging by showing the user the chronology of events, the messages exchanged between the various components involved in an application (Fig. 5).

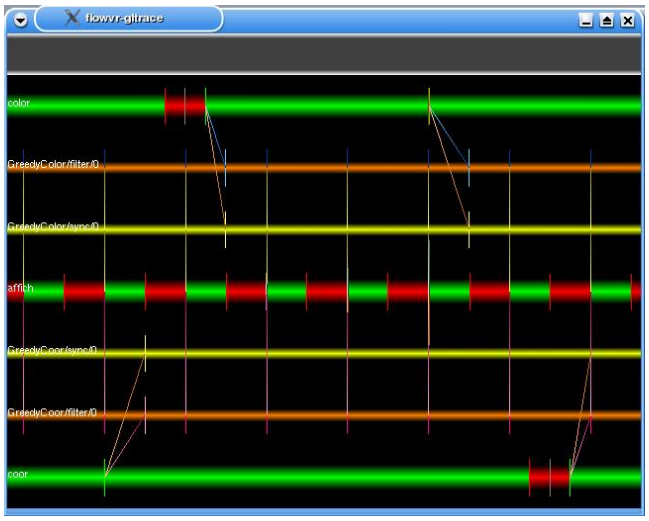


Figure 5: FlowVR OpenGL trace viewer

6 Examples of Applications

Several applications are currently built around FlowVR. They all have been developed on the GrImage platform.

6.1 GrImage Platform

GrImage (Grid and Image) is a testbed dedicated to interactive applications. GrImage aggregates commodity components for high performance video acquisition, computation and graphics rendering. Computing power is provided by a PC cluster, with some PCs dedicated to video acquisition and others to graphics rendering. A set of digital cameras enables real time video acquisition. The main goal is to rebuild in real time a 3D model of a scene shot from different view points. A display wall built around commodity video projectors provides a large, very bright and very high resolution display (about 3000x4000 pixels). The main goal is to provide a visualization space for large models and real time interactions.

Currently, GrImage is made of a 16 video projectors wall, 16 bi-opterons, 11 bi-xeons and 6 firewire cameras.

6.2 Realtime 3D Modeling

A large VR application has been developed and run on the GrImage platform. A user in front of the display wall is filmed by the cameras. The images are processed online to provide a 3D model of user that is injected into a simulation. The result is visualized on the display wall. This application enables the user to interact in real time with the virtual objects (solid objectsas well as fluids) of the application.

This application is made of about 200 modules, 5000 connections and 200 filters. It fully takes advantage of the 54 processor of Grimage.

6.3 Iso Surface Extraction

Using VTK FlowVR and FlowVR Render, we implemented an iso surface rendering from a 3D fluid simulation dataset of $132 \times 132 \times 66$ cells for 900 timesteps (one timestep is shown in figure 6).

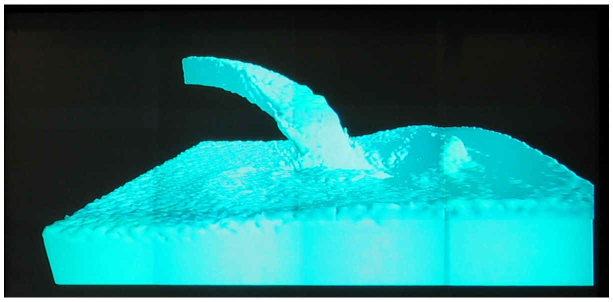


Figure 6: Iso-surface extracted from a frame of a time-varying fluid simulation dataset.

FlowVR Render outperforms Chromium [11] and shows a better scalability (Fig. 7), both while increasing the number of renderers and the number of viewers. FlowVR Render achieves 12 frames per second with 16 data viewers and 16 renderers to display the result on the 4×4 display-wall. Chromium performance is probably affected by the high overhead related to culling and stream merging operations.

6.4 Volume Rendering

We implemented a shader based volume rendering (Fig. 8) taking advantage of the massively parallel nature of today's GPUs. It is intended for the data sets that can fully be loaded in the memory of the graphics card. There are several advantage in shader use:

- Due to the massively parallel nature of todays GPUs, pixel shaders have access to more important resources, both in terms of memory bandwidth and computing power
- Shaders are able to apply transfer functions to raw volumetric data to obtain the final color and opacity. So the raw data needs to be sent only once. Only the transfert function needs to be updated.
- The use of pre-integrated [9] transfer functions and adaptive sampling steps [17] allows large datasets while keeping very high image quality.

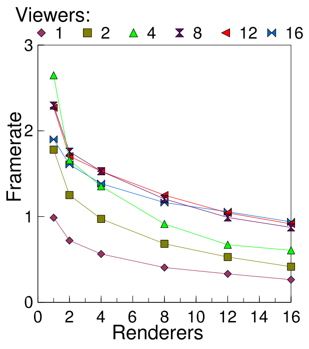
Tests were made with a Christmas tree [12] data set ($512 \times 512 \times 512$). Performance results are given in table 1. As a comparison, VTK 2D texturing implementation achieved 0.18 frames per second on one display.

6.5 Adaptive Octree Computation

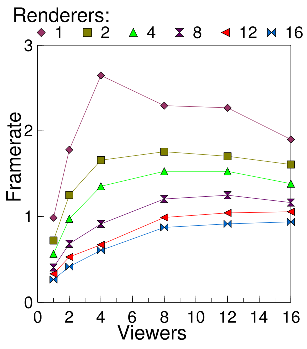
An adaptive parallel octree carving algorithm was implemented using work stealing. The octree computations are performed within a FlowVR module running on a 16 cores SMP computer. Computations are parallelized using Posix threads. Video capture and the resulting octree visualization is performed on the Grimage cluster. All the communications and flow transformations (compression...) are done by the mean of a FlowVR (Fig. 9).

7 Conclusion

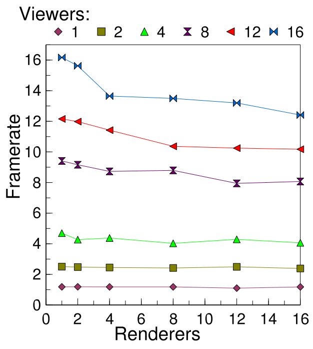
We presented the FlowVR suite. It provides a middleware for distributed interactive application favoring code reuse and modularity.



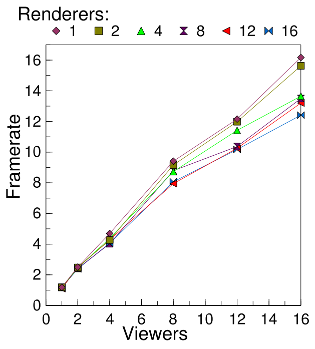
(a) Chromium



(b) Chromium



(c) FlowVR Render



(d) FlowVR Render

Figure 7: Parallel iso-surface extraction with sort-first rendering, using Chromium (a)-(b) or FlowVR Render (c)-(d). Scalability regarding the number of renderers is presented on the left, while scalability regarding the number of viewers is shown on the right.

On top of this middelware FlowVR Render defines a communication protocol for graphics primitives. It enables an efficient remote rendering on multi display environments. The suite is complemented by a video player based on Mplayer, an OpenGL wrapper and components for coupling VTK, FlowVR and FlowVR render.

In a near future, we plan to add support for automatic modules mapping on the target machine. We are also studying volume rendering of large data sets. Another project aims at connecting multiple platforms from different sites with FlowVR.

Acknowledgment

This work was partly funded by ACI project CYBER II, the RNTL project GEOBENCH, and the BULL company.

References

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

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

[3] J. Allard, V. Gouranton, E. Melin, and B. Raffin. Parallelizing Pre-rendering Computations on a Net Juggler PC Cluster. In *Immersive Projection Technology Symposium*, Orlando, USA, March 2002.

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

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

Method	Sampling steps	Framerate on 1 display Resolution 1024×768	4×4 display-wall 4096×3072	4×4 display-wall 2048×1536	4×4 display-wall 1024×768
Raycast Shader	512	1.16	2.25	5.40	8.21
Pre-Integrated Raycast Shader	512	1.10	2.04	4.97	7.70
Pre-Integrated Raycast Shader	200	2.79	5.14	12.44	19.11

Table 1: Volume rendering performances with a $512 \times 512 \times 512$ dataset.

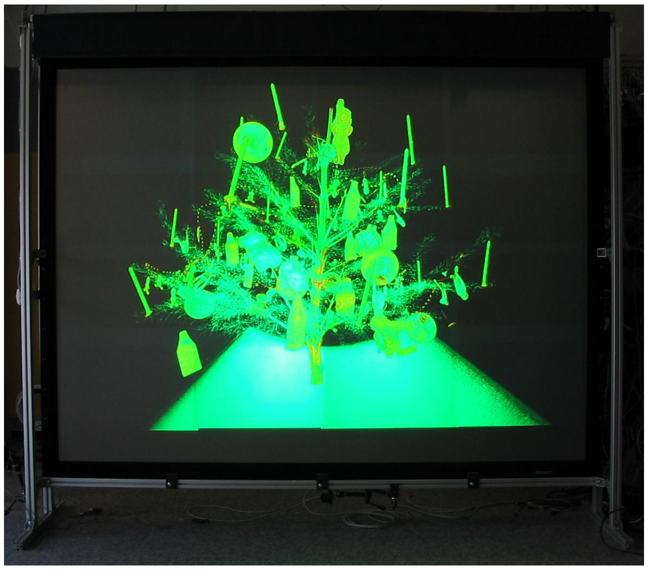


Figure 8: Volume rendering on a display wall

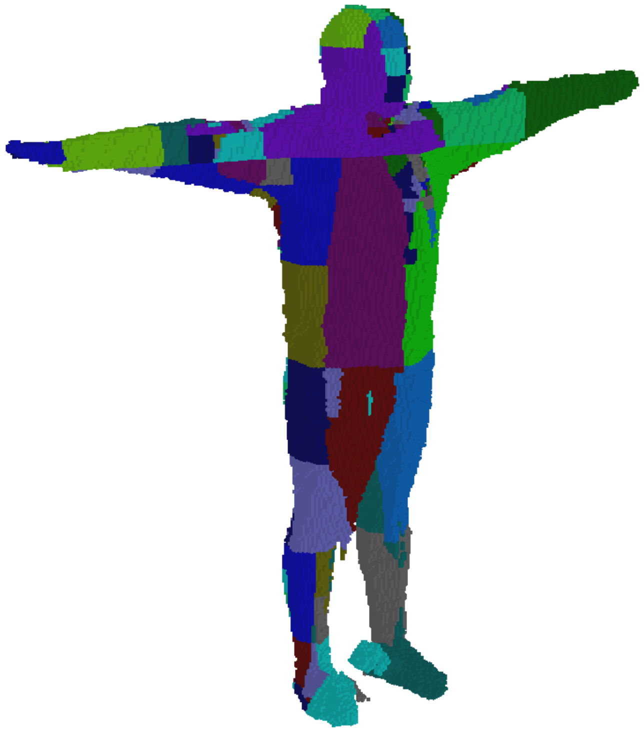


Figure 9: Example of octree computed from a set of 8 images

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

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

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

[9] K. Engel, M. Kraus, and T. Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 9–16. ACM Press, 2001.

[10] M. Gross, S. Wuermlin, M. Naef, E. Lamboray, C. Spagno, A. Kunz, E. Koller-Meier, T. Svoboda, L. V. Gool, K. S. S. Lang, A. V. Moere, and O. Staadt. Blue-C: A Spatially Immersive Display and 3D Video Portal for Telepresence. In *Proceedings of ACM SIGGRAPH 03*, San Diego, 2003.

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

[12] A. Kanitsar, T. Theussi, L. Mroz, M. Sramek, A. V. Bartroli, B. Csebfalvi, J. Hladuvka, D. Fleischmann, M. Knapp, R. Wegenkittl, P. Felkel, S. Roettger, S. Guthe, W. Purgathofer, and M. E. Groller. Christmas tree case study: computed tomography as a tool for mastering complex real world objects with applications in computer graphics. In *Proceedings of IEEE Visualization'02*, pages 489–492, 2002.

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

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

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

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

[17] S. Roettger, S. Guthe, D. Weiskopf, T. Ertl, and W. Strasser. Smart hardware-accelerated volume rendering. In *VISSYM '03: Proceedings of the symposium on Data visualisation 2003*, pages 231–238. Eurographics Association, 2003.

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

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

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

[21] S. Singhal and M. Zyda. *Networked Virtual Environments - Design and Implementation*. ACM SIGGRAPH Series. ACM Press Books, 2000.

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