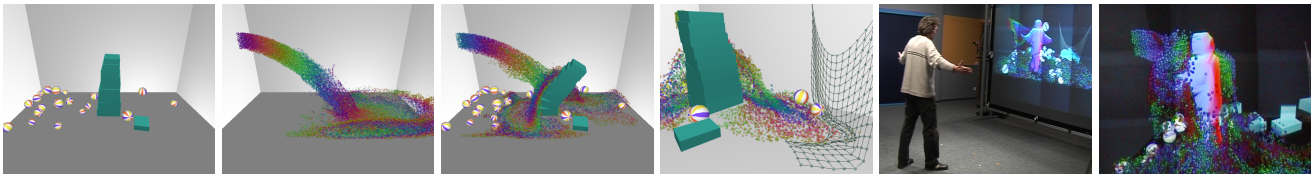
# Distributed Physical Based Simulations for Large VR Applications

Jérémie Allard* Bruno Raffin†

ID-IMAG, CNRS/INPG/INRIA/UJF
Grenoble - France



(a) (b) (c) (d) (e) (f)

Figure 1: Coupling multiple codes such as a rigid body simulation (a) and a fluid solver (b) enables to build complex worlds (c)-(d). Different distribution and parallelization approaches can next be applied to achieve real-time user interactions (e)-(f).

## Abstract

We present a novel software framework for developing highly animated virtual reality applications. Using a modular application design, our goal is to alleviate software engineering issues while yielding efficient execution on parallel machines. We target worlds involving numerous animated objects managed by physical based simulations. Mixing rigid objects, fluids, mass-spring or other deformable objects leads to complex interactions between them. Today no unified simulation algorithm with a reasonable complexity is available to manage all these types of objects.

We propose a framework for coupling and distributing existing algorithms. We reuse and extend the data-flow model where an application is built from modules exchanging data through connections. The model relies on two main classes of modules, *animators* and *interactors*. Animators are responsible for updating objects' states from forces applied to them. These forces are computed in parallel by interactors using the objects' states they receive from animators.

The network interconnecting modules can be progressively optimized. From a simple fully connected network enforcing a synchronous semantics, it can evolve towards an active network able to implement a bounding volume based dynamic routing or an asynchronous data re-sampling.

As a result, we present an application managing interactions between rigid objects, mass-spring objects and a fluid. It is executed in real-time on a 54 processors cluster driving 5 cameras and 16 projectors for user interactions.

**CR Categories:** I.3.2 [Graphics Systems]: Distributed/network graphics; I.3.6 [Methodology and Techniques]: Graphics data structures and data types; I.3.7 [Three-Dimensional Graphics and Realism]: Animation—Virtual reality; I.6.8 [Types of Simulation]: Animation—Distributed; D.1.3 [Concurrent Programming];

**Keywords:** Virtual Reality; Animations; Interactions; Physical based Simulation Coupling; Distributed and Parallel Graphics Applications; Modular Programming; Graphics Clusters;

*e-mail: Jeremie.Allard@imag.fr
†e-mail: Bruno.Raffin@imag.fr

## 1 Introduction

Developing virtual worlds that include numerous animated objects of different natures, like rigid, mass-spring, deformable or fluid objects is a challenging problem. Applications range from life-like special effects for movies, to extended possibilities of interactions in immersive virtual reality environments. For instance, virtual surgery requires to simulate a human body with rigid objects (bones), deformable objects (flesh) and fluids (blood). We can distinguish three main difficulties:

- Algorithmic issues to produce convincing animations that correctly model interactions between objects of different natures.
- Software engineering issues where multiple pieces of codes (simulations, graphics, 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, sometimes bypassed by using multiple units together (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).

Numerous approaches exist for animating objects, like keyframe, kinematic, behavioral, procedural, or physical based animations. We focus on physical based animations where object dynamics is computed from the numerical simulation of physical laws. Several algorithms have been proposed to simulate the behavior of one type of object, fluids for instance [37, 15, 14], or a collection of similar objects like rigid bodies [20]. Mixing objects of different natures, like fluids and rigid bodies, proved to be more challenging [9]. Such algorithms may require heavy computations that prevent real time executions on a single PC or workstation. To improve performance, some of them have been parallelized on PC clusters, such as interactive cloth simulation [24].

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 making efficient scene graph distribution difficult on a parallel machine [25, 21, 32]. Several scientific visualization tools adopt a data-flow model [8]. An application corresponds to an oriented graph of tasks linked by FIFO channels. 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 [36]. 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 [28, 31]. Focus was on increasing the rendering capabilities through different parallelization schemes [27]. Today, such approaches are facing difficulties to keep pace, regarding price and performance, with commodity component based platforms like graphics PC clusters [34]. But aggregating commodity components requires an extra effort on the software side. Chromium [23] proposes an 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 that copies stay coherent [12, 6, 35]. Ray tracing has also taken advantage of PC clusters to reach interactive frame rates [38]. New complex devices, like multi-camera systems [19, 26], further increase the number of components to manage and the need for parallel processing.

In this paper we present a software framework for the development of complex animated worlds enabling interactions between multiple objects, potentially of different natures. We do not propose novel simulation algorithms, but a framework to integrate existing codes. The goal is to favor application modularity in an attempt to alleviate software engineering issues while enabling efficient execution on graphics clusters.

After presenting the related works in section 2, we detail our approach in section 3. In section 5, we present our implementation, involving multiple interactions between rigid bodies, Eulerian fluids and mass-spring systems, and we detail performance results from executions on a cluster of up to 54 processors.

## 2 Related Work

Our approach builds upon prior works on physical based simulations, distributed and parallel simulations. This section gives a short overview of the background most relevant to our problem.

### 2.1 Physical Based Simulations

Simulations for computer graphics are a well studied problem, for both off-line animations and real-time interactions. The goal is to produce convincing visual effects while keeping the computational cost as low as possible, often leading to loosen the physical accuracy. Several algorithms exist to simulate the behavior of one type of object, fluids for instance, or a collection of similar objects like rigid bodies. Recent works have focused on simulations for rigid bodies [20], fluids [37, 15, 14], clothes [7], and deformable objects [13].

Implementing interactions between different types of objects, fluids and rigid bodies for instance, is a difficult issue. In certain cases this difficulty can be bypassed considering only one-way interactions, i.e. one object has an influence on another but not the reverse. A common case is when one of the object is externally controlled, either following a predetermined animation, or thru user inputs. One-way interactions also lead to sound simulations if a difference of several orders of magnitude on some parameters, weight for instance, enables to safely neglect the influence of one object onto another. Other simple approximations are sometimes used. For example, fluid to rigid body interactions can be handled using the velocity of the fluid at the center of mass of the immersed body. This can be used to compute the motion of small particles in a fluid, like leaves on the surface of a river. The most common one-way interactions are probably with immovable rigid objects. Fluid solvers generally use specific boundary conditions [14] but can sometimes handle more complex occlusions objects [22]. Cloth simulations detect and handle collisions with external objects [7].

The approximation incurred by one-way coupling may not lead to convincing visual effects. The conservation of energy requires that any energy lost by an object should be transmitted to other objects. A few recent algorithms propose to handle two-way interactions between fluids and deformable objects [17], or between fluids and rigid objects [9]. One issue is the difference in representation of each object. A rigid body uses a *Lagrangian* representation (the simulation computes properties of mobile elements), while most fluid simulations use an *Eulerian* fixed grid structure. One way to compute their interaction is by using the same model for both objects. This is the approach used in recent works, either using a Lagrangian fluid simulation [29], or adding a rigid constraint in an Eulerian fluid [9].

### 2.2 Distributed and Parallel Simulations

Networked virtual environments [36], like battlefield simulations and networked games, or distributed scene graphs [25, 21] are facing the same issue regarding data coherency. Data have to be duplicated on different machines and as soon as a value has been modified its copies should be updated. Strong coherency protocols generate a heavy network traffic and require a tight host synchronization. This is especially critical for multi-site collaborative applications where the network is an important performance bottleneck. One approach consists in loosening the consistency between copies taking advantage that the user interacts only with a sub part of the whole scene. For instance, a group of objects may be considered as a single object if located far away from the user, thus requiring only updates of the group position. Position can also be updated less frequently as the user will not notice small changes. Interpolations and data clustering techniques can be used to have hosts running at different frequencies, thus avoiding the overall simulation frequency to be bounded to its slowest host. Dead reckoning techniques have been intensively studied. They enable to extrapolate object position to absorb part of the network latency.

Scientific simulations have traditionally used parallelism to get access to high performance machines. Task parallelism is today the most common approach where computations are split into several communicating tasks. Programming relies on specific libraries like MPI [18] or OpenMP [10]. In contrast to computer graphics, the main goal is physical accuracy, usually leading to tightly coupled monolithic codes with large execution times [30]. Few works propose to reuse such approaches to parallelize computer graphics simulations to treat larger problem sizes or to reduce execution times [24]. Parallelism has been more commonly used for scientific visualization, but mainly to post-process large data sets obtained from classical parallel simulations [39, 1].

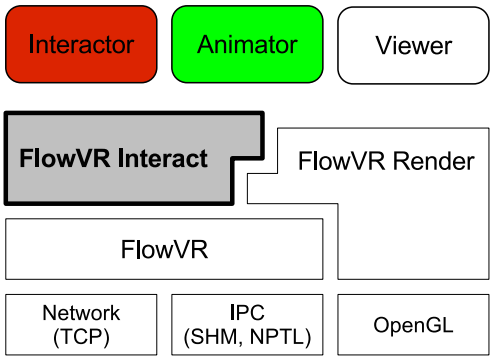



Figure 2: System architecture of our framework.

## 3 The Application Model

This section presents the model our approach is based on.

### 3.1 Overview

We introduce the main components of our framework, *FlowVR Interact* (Fig. 2). It is based on FlowVR [2], an open-source middleware for large-scale interactive applications. It is organized around an extended data-flow model. Computations take place in modules. Messages flow between modules through a network built by assembling FIFO connections and filters. Filters have the ability to perform complex treatments on messages to optimize the data flow. For rendering, a high-level extension FlowVR Render [4] allows FlowVR modules called *viewers* to describe the 3D scene in a distributed and efficient way.

The model relies on two main classes of modules, *animators* and *interactors*. 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 handling collisions between rigid objects for instance. It enables a modular coupling of existing algorithms to build worlds combining the capabilities of each of these algorithms.

### 3.2 Message Format

To allow for efficient distributed execution, the world is decomposed into independent entities identified by unique *ids*. These ids are obtained from the IP address of the host creating the object and an atomic counter stored in the host shared memory.

The world is composed of a set of *objects*. Objects are described by a set of *parameters* specific to the type of the object. Some large datasets, such as 3D meshes or vector fields, can be used by several objects. To avoid duplication in memory and within communications, this information is stored in entities called *resources*. A resource has a unique id. The objects referring to this resource only have to specify its id as a parameter.

To allow for efficient communications, only changes or events are exchanged. Each such change or event is described in a *chunk*. All data contained in a chunk are related to the same object or resource identified by its id. Messages flowing between modules are the concatenation of these chunks of data. A resource can either be static, in which case it will only be transmitted once, or dynamic, such as the mesh of deformable objects.

### 3.3 Modules, Connections and Filters

Our model is heavily based on FlowVR [2]. While we present the main concepts in this sections, please refer to [2] for more details.

A module is a computation task that can have several input and output ports. At each iteration a module waits to receive one message on each input port and it sends one message on each output port.

An application is built by selecting modules and choosing the topology of the interconnection network that defines data dependencies between modules. This network uses two components:

- Point-to-point *connections* corresponding to simple FIFO channels.
- *Filters* dedicated to message processing. In opposite to modules, a filter has access to all messages stored in the incoming buffers and can perform any operation on these messages.

By default, each module has one filter per port. On each input port, a filter, called *input filter*, gathers the last available message from each incoming connection into one message forwarded to the input port. On output ports, a filter, called *output filter*, forwards the last message received from the module to each output connection. These filters help separate network specific computations from modules.

Filters are also used for more complex message management. For instance, message broadcast can be performed through a tree of filters. The cost of broadcasting a message is thus logarithmic regarding the number of receivers if the filters can work concurrently. This cost can be further reduced using dynamic routing to remove unnecessary communications (based on bounding volumes for instance).

### 3.4 Animation Modules

A world is decomposed into *objects* distributed to animation modules, called *animators*. Each object has a unique *id* and owns a list of *parameters*. These parameters contain all data about the object that an application may need. It can include graphics data, audio data, physical data, etc. Each object has also a *type*. All objects of the same type define the same list of parameters. For instance, a rigid object animated by a physical based simulation would define a position, a bounding box, a velocity, an angular velocity, a mass, an inertia tensor, and optionally a polygonal mesh, a signed distance field, and for rendering, shaders, textures, and a (potentially more detailled) polygonal mesh.

The parameters of an object can only be updated by the animator that owns it. All objects are independent at an animator level, that is, updating a parameter of an object does not depend on the parameters of other objects. In particular, the position parameters of all objects are expressed in the same world coordinate system. This ensures that updating objects can be performed in any order, and that objects can be arbitrarily distributed to different animators. This is a fundamental property to achieve high performances in a distributed context. It also eases working only on a subset of objects.

An animator has one output port and one input port. The output port, called the *object output port*, enables an animator to send messages to notify other modules that it added, deleted an object

or updated an object parameter. At each interaction, a chunk, called object chunk, is built for each object that has changed. The message sent on the output port is the concatenation of these chunks. The input port, called the *event input port*, receives external interaction events. While the framework support several types of events, we mainly consider here forces to be applied at a specific position for a specified time duration. At each iteration, an animator updates its objects according to the forces received.

Most animators are programmed to treat a specific type of objects, thus requiring one animator for each type of objects in a scene. The objects can be further distributed by instancing several animators for the same type, partitioning the objects either statically or dynamically (based on space partitioning for instance).

### 3.5 Interaction Modules

Interaction modules, called *interactor*, handle interactions between objects. They have one input port and one output port. The input port, called the *object input port*, receives object chunks. These are the chunks computed by animators. They enable interactors to locally store the resources and objects they are concerned with. The output port, called *force output port*, sends the event chunks computed by the interactor. Different animators may be implemented to handle different interactions. For instance an animator could implement collision detection between rigid objects, while another may be dedicated to mass-spring simulations. An interactor can use the object type to select the relevant objects. Similarly to animators, the work load can be further distributed by instancing several animators, each dedicated to one region for instance.

### 3.6 Viewer Modules

Viewer modules, also called *viewers*, are in charge of transforming input data into graphical representations. Viewers usually either have one *object input port* to get object and resource data, or the animator module doubles as a viewer directly specifying the graphical representation. Viewers rely on FlowVR Render, a distributed rendering framework based on shaders we developed [4]. It enables high performance rendering on multi-projector environments.

Following the same scheme, other rendering modules could be developed, for audio rendering for instance, providing that objects define the required parameters.

### 3.7 User Inputs

User inputs can be managed in different ways, depending on their semantics. Each input device is implemented as a module. We can distinguish three cases:

- One-way user to object physical interactions:
  - The input is considered as an object, for instance a sphere centered on the tip of a pointer device. The input module is thus an animator without *event input port*.
  - The input is interpreted as a force. The input module is thus an interactor.
- Two-way physical interactions. This is typically the case for haptic devices. The input is an object that influences other objects and is submitted to other object forces. The module, an animator, considers incoming forces to compute the force feedback to apply to the user.

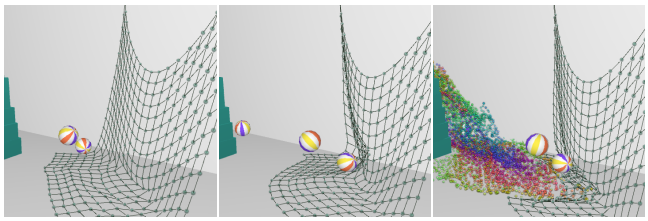


Figure 3: Animation sequence showing a mass-spring object with rigid collision then fluid interactions.

- Non physical interactions. For instance, the input device can be used to select an object, get and change the value of some internal parameters. Various alternatives are possible to implement such interactions. This usually requires the module that may respond to these interactions to support event types other than forces.

## 4 Module Assembly, Data Exchange and Filtering

We present in this section how to design an application following the model presented before.

### 4.1 Running Example

For sake of clarity, we rely on an example. This example is a simplified version of the application we implemented for our prototype. The goal is to have an animated world with rigid, fluid and mass-spring objects. Animations should respect the constraints related to the collisions that may occur between any of the objects present in the world.

We reuse known algorithms for the simulations. We rely on the algorithm of [9] to simulate the fluid and the objects that may fall into that fluid (solid and mass-spring objects). Two modifications are necessary:

- The algorithm computes the forces to apply to the objects present into the fluid, instead of directly updating the velocity of these objects (this is the role of the object animator).
- To support interactions between the fluid and mass-spring objects, we simply consider a mass-spring object as a set of independent masses. This is a simplification that for instance does not enable to control the object permeability to the fluid.

The rigid object simulation is based on [20]. The mass-spring simulation relies on a straightforward algorithm: explicit Euler integration of forces and velocity, and constant stiffness springs with maximum stretch constraint.

All objects define a position matrix, a bounding box and graphics data (mesh, textures, shaders, etc.). Each object has additional parameters depending on its type:

- Rigid objects define parameters for their velocity, angular velocity, mass, inertia tensor, polygon mesh and signed distance field.
- Eulerian fluid objects define parameters for the velocity, pressure and density field.
- Mass-Spring objects define parameters for the nodes (mass and position) and the springs (connection nodes, stiffness and damping).

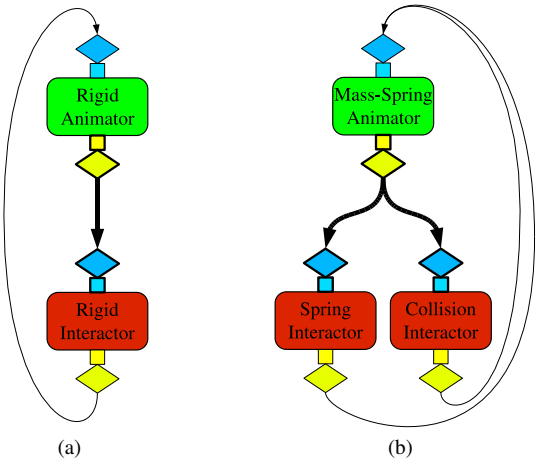



Figure 4: A rigid body simulation (a) and a mass-spring simulation (b). Ports are represented by squares, filters by rhombus. Objects description ports, filters and connections are bolded.

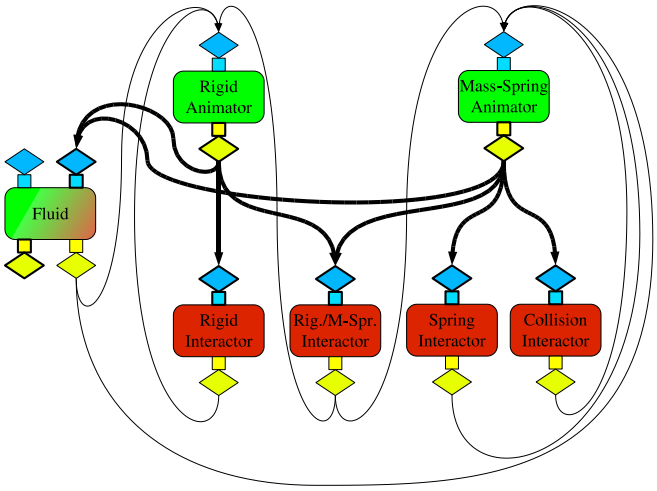



Figure 5: An Application involving two-way interactions between rigid objects, a mass-spring object and a fluid object. A first optimization of the data-flow network has lead to remove useless connections, from the spring interactor to the solid animator for instance (spring forces only have to be sent to the mass-spring animator).

### 4.2 Synchronous Application

We first start with one animator and one interactor for rigid objects (Fig. 4). A first network can be designed connecting the force ports and object ports. At each interaction, the animator updates the object states according to the collision response forces computed by the interactor.

Mass-spring objects are managed differently with an animator controlling masses position, an interactor computing forces applied by springs, an other interactor to compute forces related to collisions between mass-spring objects (Fig. 4). A third interactor is used for collisions between rigid objects and mass-spring objects (Fig. 5). The network can then be extended to connect all event ports and all object ports of all modules. Modules use the object types to discard the message chunks they are not concerned with. Animators use object ids to identify and combine forces that relate to the same object. Only one interactor managing all mass-spring related interactions could be implemented. But the modularity of the application is enhanced with 3 interactors. It enables to distribute them on different machines if extra computing power is required. It also eases testing new algorithms on one module of the simulation.

The fluid is implemented by a single module having the ports of an animator and an interactor module. This animator/interactor manages the fluids dynamics taking into account the objects that can be present into the fluid. The output forces are the forces resulting from the fluid pressure on the other objects. The algorithm [9] is based on a 3D grid discretizing the space area where fluid can be present. Separating the animator from the interactor would not be appropriate, as this algorithm uses large internal data (velocity and pressure fields, marker particles), and forces for the fluid itself as well as the immersed objects are computed simultaneously. As we will details in section 4.3.2, we propose to parallelize the module following a different approach that enables to achieve more significant speed-ups.

The network designed includes cycles, for instance between the animator and the interactor of the rigid objects. These cycles cause deadlocks that prevent the application to start. We rely on input filters to generate at start-up a default message that enables their modules to start. If only the animators' input filters create this initial message, then the animators and interactors will execute successively, similarly to the classical sequential simulation loop. To allow all modules to run in parallel, all inputs filters must create one or more initial message. This will increase performances but at the expense of the latency of the interactions, as the animators are now receiving forces computed using objects' states several iterations old. This is mostly an issue for rigid interactions, such as collisions, and less noticeable for smooth interactions, related to the fluid of the springs for instance. In this case, we can set the input filter of the rigid interactor not to send an initial message, forcing the rigid animator and interactor to run sequentially.

By default, modules are fully interconnected. A simple optimization consists in removing useless connections (Fig. 5). Without further effort, off-line animations are computed executing all modules on the same machine. Figure 3 shows an excerpt of the resulting animation.

### 4.3 Distributed Application

Several optimizations can be implemented to improve performances. A simple first step is to distribute the modules on the hosts of a cluster to benefit from extra computing power. The synchronous semantics of the network ensures data dependencies are respected.

#### 4.3.1 Module Duplication

Another optimization consists in reducing the work load of a module by running several copies in parallel on different hosts. This approach does not require to modify the module code if its workload can be divided into independent batches. For instance such approach can be applied to run several copies of the rigid interactor, each managing a different volume in space.

#### 4.3.2 Parallel Module

Duplication may not be possible for all modules. For these modules a classical parallelization can be considered. The fluid module

implements a simulation based on a 3D grid. A classical parallelization scheme for such data structure is to partition the 3D grid into blocks, distribute the blocks to different processors and exchange the border values between neighbors at each step. This parallelization is efficient for large datasets as the communication cost is proportional to the block surface, while the computation cost is related to the block volume. Such parallelization is easily implemented with MPI [18] for instance. In this case, we implement one module per block. The network must be modified to ensure input data are properly distributed to the modules and output data are properly gathered. Filters are used to perform these operations.

#### 4.3.3 Dynamic Routing

On clusters the network is often the bottleneck. Care must be taken to avoid sending useless data. Static optimizations are possible (Fig. 5), but filters can also be programmed to dynamically route messages. Assume we use several interactors to compute collisions. The list of objects that are relevant to each interactor depends on the space partition it is managing. An additional output port is declared on each viewer to send their volume of interest to the output filters of the animators. The filters are modified to forward to each viewer only the chunks of the objects whose bounding volume intersects with the volume of interest. These filters must also save the state of all objects to forward interactors up-to-date parameters when needed.

#### 4.3.4 Multi-rate Application

The application and all its variations afore-mentioned are synchronous, i.e. all modules run at the same rate, the rate of the slowest module. Performance can be significantly improved having modules running at different rates. Filters can be used to implement multiple fixed rates.

Let us consider the mass-spring simulation (Fig. 4(b)). It is based on a very simple algorithm that is unstable if the time step is too large. However the time step is defined by the slowest module in the scene, the fluid in our case. To have the mass-spring simulation running 10 times faster than the other modules, the input filters are modified. For each event chunk received, the filter forwards 10 consecutive messages with the same data but a time duration divided by 10. The mass-spring animator thus executes 10 iterations while other animators iterate only once. Every 10 messages created by the mass-spring animator, the output filter forwards only the most recent one to the other interactors in the application. This scheme allows to use different time steps within the application, transparently from the point of view of the animators and interactors.

## 5 Implementation and Results

This section introduces our prototype implementation and discusses performance results based on an application involving rigid, mass-spring and fluid objects.

### 5.1 FlowVR

Our implementation relies on FlowVR[2]. The list of modules selected for an application and the network (connections and filters) are described with Perl scripts. Predefined functions implement common patterns. They can be extended with new functions. In our case for instance, we developed a new function to automatically associate with each module its input and output filter. This approach enables to describe an application in a compact way even if the resulting network is very complex. This script also enables to map each module or filter onto a given host when the application is executed on a cluster.

FlowVR relies on a daemon executed on each host of the cluster. The daemon manages a shared memory segment. All messages created by a filter or module are allocated in this segment. When modules or filters located on the same host exchange a message, it results in a simple pointer exchange, thus saving costly message copies. Inter-hosts message exchanges are transparently handled by daemons that perform the required data transfers.

### 5.2 Results

The application implemented is the one described in section 4.

#### 5.2.1 Single Machine Execution

We first consider an execution on a single machine without user interactions. A FIFO network ensures that all modules run at the same rate. The application was tested with the following objects:

- Rigid body simulation, with 20 objects
- Parallel fluid simulation on a $32 \times 64 \times 32$ grid.
- A mass-spring 2D net, with $20 \times 20$ nodes.

The application was executed on a 1.6 GHz dual Opteron PC equipped with NVIDIA FX 5700 graphics cards. The application reached 6.5 frames per second (fps), which is not sufficient for virtual reality. The attached video enables to evaluate the visual quality of the animations.

#### 5.2.2 Interactive Execution

We next targeted real-time executions including advanced user interactions in a semi-immersive environment.

The application was executed on the GrImage platform that consists in a cluster of sixteen 1.6 GHz dual Opteron PCs equipped with NVIDIA FX 5700 graphics cards and connected together by a gigabit Ethernet network.

The rigid-fluid simulation being the slowest module, it is parallelized with MPI [18] following the approach presented in section 4.3.2. Messages from and to the modules of this parallelization transit through filters organized into a binary-tree. This parallel simulation was executed on 8 hosts. The remaining modules were distributed one per host.

The application was modified to support multiple rendering hosts, driving a display wall of sixteen projectors. The display was synchronized using a software swaplock to ensure a proper image synchronization (implemented with FlowVR Render). The data from the mass-spring and rigid animators was simply broadcasted to all rendering hosts. The fluid however generates a large set of particles (up to 200000 particles) that are used for rendering. To avoid collapsing the network, the output filter of each module of the parallelized fluid was set to dynamically route the grid cell content according to each projector's frustum.

For user interactions we additionally used 5 FireWire cameras connected to 11 dual-Xeon 2.6 GHz PCs computing a user's 3D model

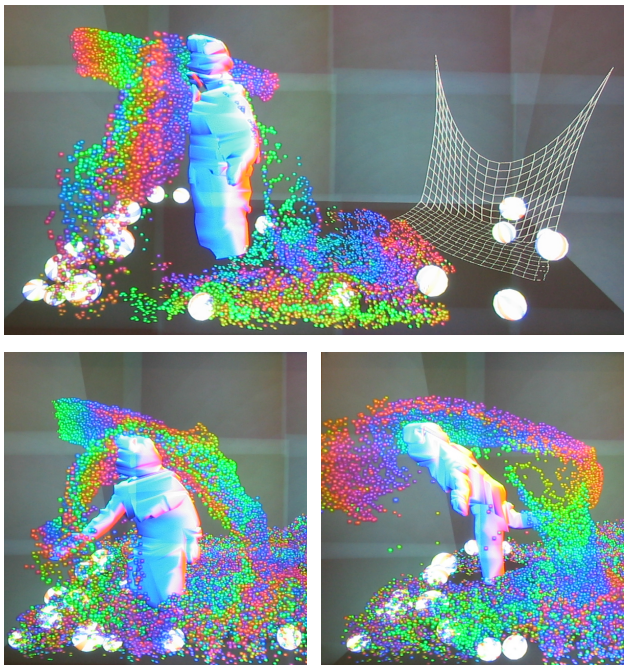


Figure 6: Interactive execution with camera-based user interactions and high resolution rendering on a display wall.

in real-time. We added a new animator dedicated to the user's model. This model is considered as a rigid object. It defines all required parameters. Note however that 3D mesh and the signed distance field are computed dynamically:

- The 3D mesh is obtained from the computation of an exact visual hull [16].
- The signed distance field is computed by applying an Euclidean distance transformation [33] from a voxel model of the user [11].

These parameters provide only position data and no velocity, which limits the possibilities of interactions with other objects. Regarding interactions, this object was managed as a classical object except that no force affects it (it can be considered as an infinite mass body).

The animator has its output object port connected to the input object port of the rigid, rigid-fluid and rigid/mass-spring interactors. It receives on an extra input port the 3D mesh computed in parallel based on the algorithm of [16]. An other module computes the voxel model and the signed distance field [3]. Pictures of the rendering on the display wall is shown in Fig. 6.

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. Notice that the network is an important bottleneck. When all fluid particles are forwarded to all 16 rendering hosts, the frame rate is below 1fps. The dynamic routing we use enables to significantly reduce this bottleneck to reach 18 fps. In the future, the fluid modules will be modified to extract and forward only the fluid surface instead of particles.

## 6 Conclusion

We presented a software framework for coupling physical based simulations for large VR applications. Our approach relies on objects distributed amongst animators in charge of updating the objects' state, and interactors computing the forces that apply to objects. This approach enables both a modular application development as well as an efficient distributed execution. Objects being self-defined, it enables to distribute them amongst different animators. The different forces that apply to an object can be computed independently. It allows to have several interactors working in parallel.

We presented an application coupling a rigid-fluid simulation, a rigid bodies simulation and a mass-spring simulation. It reached interactive frame rates by distributing the modules on a cluster and using a classical grid-based parallelization of the rigid-fluid simulation. Complex devices, a multi camera system and a display wall, were attached to the application to allow user interactions. It demonstrated the effectiveness of modularity and the interest in using a cluster to enable highly animated VR applications.

Our approach takes advantage of the underlying dataflow model where connections can be combined with filters to enable complex message processing. It clearly separates module programming from network programming. The user can concentrate on implementing algorithms into modules first and design a simple initial network for testing purpose. Next, he can focus on performances by distributing modules on a cluster. The dataflow network can then be gradually improved to implement different optimizations, like dynamics routing or multi-rate execution. Heavily loaded modules may be duplicated or parallelized to further improve the frame rate and the latency.

Tightly coupled simulation algorithms do not adapt well to the animator/interactor model. They rely on strong data dependencies that require specific parallelizations to achieve high performance executions. The modular design we propose enables to use such scheme when required and switch to more efficient algorithms progressively as efficient implementations become available.

Future works will address long range interactions between objects, like light interactions. The goal will be to distribute the shadow computations into several specialized modules. Experiments will also be conducted to develop large scale applications taking advantage of 100 to 200 processors.

## References


[1] J. Ahrens, C. Law, W. Schroeder, K. Martin, and M. Papka. A Parallel Approach for Efficiently Visualizing Extremely Large, Time-Varying Datasets. http://www.acl.lanl.gov/Viz/papers/pvtk/pvtkpreprint/.

[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 *Proceedings of Euro-par 2004*, Pisa, Italia, August 2004.

[3] J. Allard, C. Ménier, E. Boyer, and B. Raffin. Running large vr applications on a PC cluster: the FlowVR experience. In *Proceedings of EGVE/IPT 05*, Denmark, October 2005.

[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. Rühle. Architectures of Distributed Visualization Systems and their Enhancements. In *Eurographics Workshop on Visualization in Scientific Computing*, Abingdon, 1993.

[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] R. Bridson, R. Fedkiw, and J. Anderson. Robust treatment of collisions, contact and friction for cloth animation. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 594–603. ACM Press, 2002.

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

[9] M. Carlson, P. J. Mucha, and G. Turk. Rigid fluid: animating the interplay between rigid bodies and fluid. *ACM Trans. Graph. (Proceedings of ACM SIGGRAPH 04)*, 23(3):377–384, 2004.

[10] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, 2000.

[11] G. Cheung, T. Kanade, J.-Y. Bouguet, and M. Holler. A real time system for robust 3d voxel reconstruction of human motions. In *Proceedings of CVPR'00*, volume 2, pages 714 – 720, June 2000.

[12] 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.

[13] G. Debunne, M. Desbrun, M.-P. Cani, and A. H. Barr. Dynamic real-time deformations using space and time adaptive sampling. In *Proceedings of ACM SIGGRAPH 01*, Annual Conference Series. ACM Press / ACM SIGGRAPH, August 2001. Proceedings of ACM SIGGRAPH 01.

[14] D. Enright, S. Marschner, and R. Fedkiw. Animation and rendering of complex water surfaces. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 736–744. ACM Press, 2002.

[15] N. Foster and R. Fedkiw. Practical animation of liquids. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 23–30. ACM Press, 2001.

[16] J.-S. Franco, C. Ménier, E. Boyer, and B. Raffin. A distributed approach for real time 3d modeling. In *Proceedings of the IEEE Workshop on Real Time 3D Sensors and Their Use*, Washington, USA, July 2004.

[17] O. Génevaux, A. Habibi, and J.-M. Dischler. Simulating fluid-solid interaction. In *Graphics Interface*, pages 31–38. CIPS, Canadian Human-Computer Commnication Society, A K Peters, June 2003. ISBN 1-56881-207-8, ISSN 0713-5424.

[18] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. Scientific and Engeneering Computation Series. The MIT Press, 1994.

[19] 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.

[20] E. Guendelman, R. Bridson, and R. Fedkiw. Nonconvex rigid bodies with stacking. *ACM Trans. Graph. (Proceedings of ACM SIGGRAPH 03)*, 22(3):871–878, 2003.

[21] G. Hesina, D. Schmalsteig, A. Fuhrmann, and W. Purgathofer. Distributed Open Inventor: A practical Approach to Distributed 3D graphics. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*, pages 74–81, London, December 1999.

[22] B. Houston, C. Bond, and M. Wiebe. A unified approach for modeling complex occlusions in fluid simulations. In *Proceedings of the ACM SIGGRAPH 03 conference on Sketches & applications*, pages 1–1. ACM Press, 2003.

[23] 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.

[24] M. Keckeisen and W. Blochinger. Parallel Implicit Integration for Cloth Animations on Distributed Memory Architectures. In *Proceedings of the Eurographics Parallel Graphics and Visualization Symposium*, Grenoble, France, June 2004.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] M. Müller, S. Schirm, M. Teschner, B. Heidelberger, and M. Gross. Interaction of fluids with deformable solids. *Journal of Computer Animation and Virtual Worlds (CAVW)*, 15:159–171, 2004.

[30] T. Pohl, N. Thuerey, F. Deserno, U. Ruede, P. Lammers, G. Wellein, and T. Zeiser. Performance Evaluation of Parallel Large-Scale Lattice Boltzmann Applications on Three Supercomputing Architectures. In *Proceedings of SC2004*, November 2004.

[31] J. Rohlf and J. Helman. IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics. In *Computer Graphics (ACM SIGGRAPH 94)*, pages 381–394. ACM Press, July 1994.

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

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

[34] 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.

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

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

[37] J. Stam. Interacting with smoke and fire in real time. *Communications of the ACM*, 43(7):76–83, 2000.

[38] I. Wald, C. Benthin, and P. Slusallek. Distributed Interactive Ray Tracing of Dynamic Scenes. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG)*, 2003.

[39] C. Wang, J. Gao, and H.-W. Shen. Parallel Multiresolution Volume Rendering of Large Data Sets with Error-Guided Load Balancing. In *Proceedings of the Eurographics Parallel Graphics and Visualization Symposium*, pages 23–30, Grenoble, France, June 2004.