Proceedings of Virtual Reality International Conference (VRIC 2010), 7-9 April 2010, Laval, France.
RICHIR Simon, SHIRAI Akihiko Editors. International conference organized by Laval Virtual.

Toward distant interactive VR Visualization of Large Parallel Simulations

Arvaux Simon[1], Melin Emmanuel[2], Robert Sophie[3]

Laboratoire d'Informatique Fondamentale d'Orléans, Université d'Orléans
Orléans, France
[1] simon.arvaux@univ-orleans.fr
[2] emmanuel.melin@univ-orleans.fr
[3] sophie.robert@univ-orleans.fr

Abstract—**In this paper we present a computational steering application framework, providing distant visualization of multiple simulations. The FlowVR virtual reality framework is used to handle both code coupling and resulting synchronization concerns. We discuss synchronization problems caused by the integration of a steerable heavy simulation, and present a possible solution addressing both latency imperatives and necessity of optimizing the simulation process. As we target high performance visualization of large data, PC clusters are used to allow in-core treatment of simulation results. This is requiring parallelization of the LOD algorithm implied, aiming at simplifying simulations output for transmission over the network once data compression is accomplished. As an implementation experimentation, we present integration of a MPI parallelized dynamic flood simulation in a static terrain data distant visualization application, both simulations allowing asynchronous computational steering across the network.**

Keywords-component; VR; Simulation; HPC; code coupling; Flooding

I. INTRODUCTION

Since they involve large scale, 3D and multiple screens, VR environments are very good candidates to display high amount of dynamic data issued from heavy simulations or even huge static one issued from laser scanning like *Light Detection and Ranging* (LIDAR). Moreover VR offers enriched interface to interact with data and possibly to steer heavy simulations. Simulations involving great amount of data and lots of parameters are known to be very difficult to tune since their parameters or input data may have to be slightly modified and linked to intermediate simulation results. Thanks to their rich interfaces, VR environments are perfect in order to browse through static and dynamic data, understand phenomena, modify input data to see logical relations, and steer computation to make a good use of computing power.

VR environment are very power consuming, therefore they are now driven by parallel architecture but they cannot compute heavy simulations since the computing power needed by these simulations may be more than one order of magnitude superior. Moreover, various data necessary to such simulations may be located on different places in the world and might have access restrictions making impossible their transfer to other locations. For all of these reasons, it is useful to launch large distant simulations taking the most of non local power computing and datasets of Grid architecture. In this case, VR environment appears as a powerful local visualization and interaction client. On top of that, users of the application may be multiple and above all not on the same sites. For such reasons, we can imagine several VR environments sharing and steering altogether shared distant simulations.

Due to intrinsic hardware limitations on SMP architectures, only distributed parallel architectures, like clusters, have the power to run heavy parallel simulation codes. In this case, the classic programming model is SPMD with the MPI library as message passing interface. This model is well suited to monolithic big simulations but much less adapted to code coupling of several different simulations. Code coupling frameworks exist but are generally not conceived for interactive code. This is the reason why we propose to adapt FlowVR, a code coupling framework dedicated to high performance VR.

The FlowVR architecture is based upon the concepts of modules (pieces of software performing a treatment using I/O data) and the FlowVR network, which is necessary to assemble the modules into a working application. A FlowVR module may be a MPI parallel code itself. The FlowVR network manages the semantic of connections and synchronizations between modules via

a library of network objects. Due to the fact that FlowVR is made for real time interactive VR applications, the semantic of network objects is oriented to asynchronism, sometime at the price of a quite big overhead for the hardware.

In this paper we propose a general framework to conceive a FlowVR application dedicated to visualization and navigation on multiple sources of huge dataset coming from distant parallel simulations. We offer the capability to steer the simulation in two different ways. First we modify, via the VR environment, data inputs of the simulation. Second, we propose a new FlowVR network object, taking into account semantic of data coming from FlowVR modules to over-synchronize modules, saving computing power on local and distant parallel resources.

We first present our general framework, next we describe our implementation. For this example, we have chosen a flood simulation. Floods have caused important damage in several countries across the world and their prevention to limit human and material costs is a requested search result. Simulating a flood prediction using computers is an interesting way to make the best prevention possible. An important computation power is necessary to simulate quickly the consequences of a flood predicted in a near future, targeting accuracy and realistic behavior of the simulation on top of that. As a result we provide a generic architecture to visualize a geographical zone and coupling any simulation over this kind of application. Our goal is to introduce computational steering concerns inside an existing VR framework, and develop new synchronization methods to address performance issues. The described application has been designed with scientific visualization of simulations results in mind, thus we use a VR client to provide visualization of heavily parallelized server side simulations.

II. General framework

A. Key concepts

We're interested in several key concepts to provide an application able to answer our requirements:

- Distant visualization through VR environment;
- Heavily parallel simulation codes based on MPI and steered via VR environment;
- Code coupling of these heterogeneous codes and fine management of synchronization policies inside a logic network.

B. Framework

We now describe our general architecture. Grid computing aims at combining multiple computer resources and apply them to a common scientific task. When large amounts of data needs to be processed using a great number of computer processing cycles, and the results may be viewed on various distant sites, Grid computing is the way to go. In this work we take advantage of the Grid concept by exporting calculations on distant sites for example from where input data are stored. Large MPI simulations may be computed but classically their outputs are large data sets impossible to export via *wide-area networks* (*wan*). Therefore a key concept of our logical architecture is the LOD modules allowing to simplify data in such a way that it is now possible to transfer them through the *wan* and rebuild it without losing too much of relevant information. Our framework is summed up in fig 1. We create a FlowVR network embedding several simulations. Data issued from them are treated by a FlowVR compressor module. This module compresses data via a LOD algorithm. Then data are transmitted over the network. Data are uncompressed thanks to a LOD FlowVR module and then sent to the visualization FlowVR module which merges all data received from all simulations. Note that static data follows a similar process.

The problem we want to solve is the following : starting from an existing static data distant visualization code, how to integrate one or more dynamic simulations possibly relying on these static data ? Moreover, as the simulation may be steered by the viewer, what synchronization policy should we use ? In the case of static data visualization, only the compressor needs to be controlled and slowed down when the viewer isn't moving, but concerning simulations, they may not only be steered by the client, but also update their results to the viewer periodically. The FlowVR model solves the code coupling aspect and also partially takes care of the synchronization required. The FlowVR model is very versatile, so it is possible to enrich it by implementing new network component so as to plainly fulfill optimization concerns imposed by network latency.

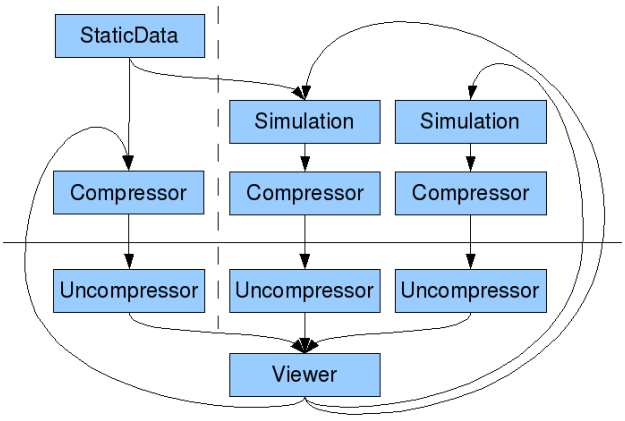

Fig 1: General framework

C. *Computational steering*

Computational steering is a recent and important concept in the scientific research domain. This discipline aims at improving the computational process and scientific analysis by introducing interaction within a simulation. Using this approach, the user is no more waiting passively for the results of the computation, on the contrary he can visualize in real time calculus evolution. Furthermore, the user is able to interact anytime with the model and modify simulation parameters on the fly in order to steer the course of the computation. Using several previous works on computational steering, A. Esnard underlines in his PhD [1] thesis key characteristics : firstly, interaction is done in real time, while the simulation is ran. Next, the interaction should remain coherent regarding the simulation, it must be kept connected to the underlying physic model. Finally, the interaction should be effective in a sense that it should not introduce perturbations by slowing down the simulation.

D. *Parallelism*

In our framework, we use a FlowVR based generic architecture with modules involving MPI parallelized simulation. MPI is a protocol to handle communications between processes in parallel computers development. Described as “a message-passing application programmer interface, together with protocol and semantic specifications for how its features must behave in any implementation.” by Gropp [2], it remains a dominant model concerning high performance computing nowadays.

FlowVR is a framework whose purpose is to design easily applications combining both Virtual Reality and parallelism [3]. The FlowVR architecture is based upon the concepts of software components: modules following a wait/get/put pattern meaning they wait until each of their input port is fed with new data, then they retrieve this data to compute their results finally sent on the output ports. The FlowVR network is responsible for interconnecting modules composing an application using connections between their I/O ports.

Integration of an existing code inside a FlowVR application doesn't require a complete rewrite as the FlowVR framework implements the software components concept, it is only necessary to interface input and output handling with the FlowVR framework API, the computational core needs no modifications. Software components is ideal to allow code coupling integration, as it is easy to inject existing codes inside a FlowVR network, using the framework API to glue heterogeneous codes together.

FlowVR also provides solutions to deal with the synchronization of the different modules across the network. *FIFO* is the default mode for FlowVR connections if no synchronizer is controlling it. All messages sent by the producer will be consumed in the same order by the receiver. Respecting the FIFO model, no messages will be lost or inverted between the components. As a direct consequence, all the modules connected in a FIFO cyclic connection will run at the exact same speed of the slowest module, so it is used to connect totally synchronized modules. On the contrary, there is a completely asynchronous connection known as the *Greedy*.

The greedy (or sampling) mode allows modules to periodically read incoming messages whose data is asynchronously updated. Designed specifically for low latency and real time applications, each software component can take advantage of this synchronization method to run at optimal speed. At the beginning of each new iteration, the consumer will use the latest data available and ignore previous updates, discarded by the connection.

The key objects to implement this type of connection between two given modules are the GreedySynchronizor synchronizer and the FilterIt filter, as shown on this diagram :

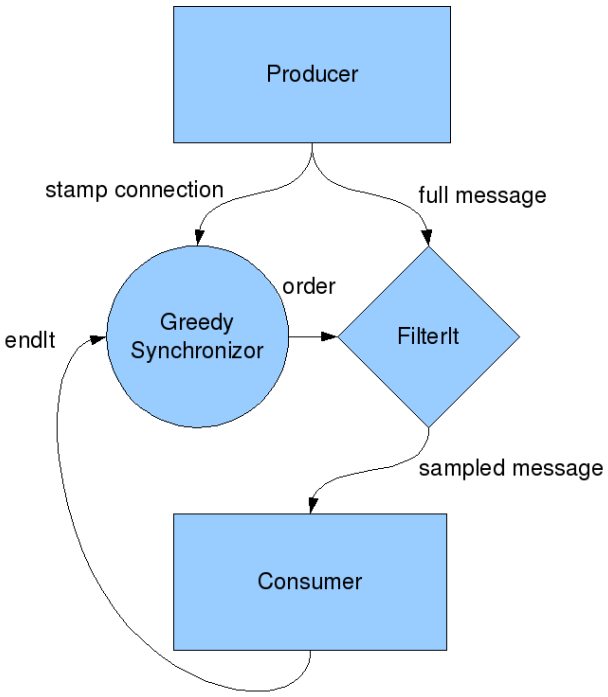

Fig 2: Greedy connection

As shown in fig.2, the synchronizer has two input ports : stamps, which should be connected to the source of the data (producer module) and endIt, receiving the signal that the consumer module has finished an iteration through its own endIt activation output port. Using this information, the synchronizer is aware of the precise

moment when the receiver has finished handling previous data and is thus requesting new one. To do so, it has to read the stamps (meta data associated with a message) received on the corresponding port to determine the most recent message. These chosen stamps are then forwarded to the order output port, connected to the corresponding order input port on the filter. The filter has another data input port named *in*, which should be connected to the same source of data (emitter) than the synchronizer is. Finally, the filter will use information collected about message ordering to decide in which order the messages should be forwarded (or sampled) to the destination module.

Unfortunately these standard synchronization policies are insufficient considering our objectives: steering of heavy simulations running server side and distant visualization of the results using a non local client. To optimize CPU power on the server we want to launch simplification and compression of static data only when necessary (for example when camera was moved by the user). Note that this management of computations has another important benefit: it optimizes network load since simulation results are only sent when necessary. As a consequence we investigate the need of a new type of connection to answer these needs.

E. Steer Greedy connections

We introduce a new type of FlowVR network object : the SteerGreedy connection. It is a slight variation over the Greedy connection, named after the concept of monitoring the connection and the modules connected.

The SteerGreedy connection should provide the exact same semantic as a Greedy connection while the messages produced by the emitter module are interesting for the consumer module point of view (meaning these messages are always different thus triggering new computations server-side). However when the emitter decides his messages are no longer interesting, he should give a hint to the synchronizer in order to block the connection and pause the receiver. The receiver module, not fed with an incoming message, would then wait actively until the connection is enabled again, such behavior being much less CPU consuming than useless computations. The consumer module is unblocked whenever the synchronizer receives a new message not already sent over the connection.

Furthermore, the SteerGreedy connection semantic must allow a chain reaction to block and unblock modules connected together with multiple SteerGreedy connections. Inter blocking is avoided using a standard Greedy (always non-blocking and asynchronous) in at least one point in the connection loop. The SteerGreedy synchronizer must be informed using meta-data on the messages sent (stamps) about the pertinence of their content. The degree of pertinence of a message, conditioning its perception to be sent as a new message over the connection, could be further parametrized but for now on, we decide that a message is pertinent as long as it is totally different from its direct ancestor in the message queue. This is why the producer module must always compute additional meta data along with its output messages, as it will be used to account for their relevance later in the SteerGreedy connection.

Let's explain more precisely the SteerGreedy principle, by reviewing a use case where we establish a connection between a producer and a consumer module. The producer module computes iteratively output messages, with associated codes corresponding to a unique identifier of the message data. Then this code is transmitted to the synchronizer as a stamp meta data information, while the full message data is sent to a filter. The filter role is simply to store the most recent copy of the messages produced by the emitter, that is to say the last sent message. At the same time, the synchronizer compares the code of the last message produced with the code of the last message emitted to the consumer, it is able to do so because a copy of the code representing the last emitted message by the filter is stored locally. If the two codes match then the connection is blocked until further notice (reception of a new message, coded differently by definition), but if they are not the same, then the synchronizer replaces the stored code by the new one and informs the filter that it is time to emit a new message. Of course this process takes place only if the consumer module has requested fresh input to the connection by emitting on the “endIt” port, connected to the synchronizer.

III. Framework instantiation

A. Application presentation

We start from an application allowing distant visualization of large static data structures, using a LOD algorithm [4]. The data on the server are not handled using a classic out-of-core approach, in this case, height fields are loaded in RAM using PC clusters in a parallel environment. Using these techniques and characteristics, this application can handle, not only static, but two, dynamic simulations if we consider a computation and resources overload. Our main goal is to enable steering of heavily parallel simulation codes coupled each other, and being able to perceive the results on a distant visualization client using virtual reality. For that matter, we propose to visualize the results of a flood simulation on large height fields data. The results computed on the server are in both case simplified using a LOD algorithm and compressed for emission on the network, as further detailed. Notice that the flood simulation happens on every needed cell, not only the visible ones.

As the simulation code is coupled to the visualization one, it is possible to change the behavior of the simulation without changing its code, the modifications are propagated thanks to the FlowVR network. To achieve this we take advantage of the coupling model and capabilities of FlowVR in order to develop unsynchronized modules and minimize the code modifications needed.

B. Distant visualization of large static data

The possibility to visualize and interact with data locally unavailable is an interesting feature but it requires an important prerequisite : minimizing the amount of data transmitted through the distant modules to avoid network saturation. Moreover these data are supposed to be quite large so they must be compressed by two different means :

- First we apply a LOD algorithm on the geometry sent to the client to enable visualization of very large scene.

- Then we apply a compression on the degraded scene geometry structure to further minimize the data sent over the network.

More precisely, we rely on the CLOD (Continuous Level Of Detail) algorithm to compute an approximation of the current scene geometry simplified according not only to the camera point of view, but also terrain roughness. The idea behind this process is that when it comes to visualizing an height field, flat areas should be oversimplified while mountainous zones must keep their precision to look like refined data. This algorithm decomposes the terrain into quadtrees (tree with four children) : from the initial terrain cell, we divide it into 4 parts then start again until we reach the maximal refinement level or the one needed (depending on the distance from the camera and terrain roughness). Each node in the data structure hierarchy corresponds to a level of detail, quadtrees are then rendered using triangles. The deeper the level of detail, the smaller the triangles hence leading to more precision of the detailed zone. A preprocessing phase must be performed in order to compute D2 constants for each node until the maximum level of detail is reached, these constants, representing the terrain roughness degree, are then organized in a similar quadtree structure for easier assimilation to their respective node. The CLOD algorithm then consists of traversing recursively the quadtree structure and make a decision at each node whether it needs further refinement or not.

The resulting quadtree structure is then compressed using the SSQ (Simplified Sequential Quadtree) principle. The hierarchy and structure of a quadtree can be encoded in one single byte : the first 4 bits indicating if it has a child at a given position, and the following 4 bits standing for the presence of an edge on each 4 sides. The quadtree must have an edge if it has a neighbor having the same level of detail as itself. As a result, a quadtree structure can be compressed as only an array of bytes and the corresponding array of node values. S. Madougou provided these results in [4] and we use them to implement a parallel version of the CLOD and quadtree compression algorithm. During a pretreatment phase, we divided the data set in cells of equal size usable by the CLOD algorithm (2^n+1, this is inherent to the quadtree decomposition process). Then these cells are distributed among parallel modules on the server, calculations are realized in parallel and results gathered and synchronized for visualization on the distant client.

C. Integration of a heavy simulation distantly steered

1) Flood simulation choice

As far as flood simulations are concerned, existing literature provides many clues and directions in order to help deciding how to chose a correct implementation. In 1999, Stam described the "stable fluids" method [5] to implement fluid simulations using a computational fluid dynamics based approach. This article tackled many aspects of computational fluid dynamics (a branch of fluid mechanics using numerical methods and algorithms to solve and analyze problems involving fluid flows). Further articles [6] helped vulgarization of the method and optimization of implementation. In computational fluid dynamics, HPC are used to perform the millions of computations required to simulate the interaction of liquids and gases with surfaces defined by boundary conditions. Even with such CPU high speed power, in many cases only approximate solutions can be achieved.

The fundamental basis of almost all CFD problems are the Navier-Stokes equations, which define any single-phase fluid flow. These complex equations can be linearized and simplified by removing terms corresponding to parameters we're not interested in concerning water flow simulations. Continuous fluids must be discretized before being representable in computer memory, that's why we chose to discretize the spatial domain into small cells to form a volume grid (finite volume discretization method), and then apply a suitable algorithm to solve the Navier-Stokes equations.

2) Actual implementation

However for ease of implementation purpose, we chose a rather naive algorithm to control the flood simulation. Nonetheless, the simulation module I/O is clearly formalized and generic so replacing the module by a more realistic simulation is straightforward. The simulation is parallelized thanks to MPI communications : modules need to exchange their borders values before performing each iteration step. Details on general parallelization of flood simulations for clusters using MPI can be found in [7].

The characteristics of the simulation are as follow :

- It runs asynchronously alongside the geometry simplification process which is coupled with.
- It supports alteration of the topography and modification of its parameters (water sources intensities...) via client user input.
- Key steps of the simulation can be saved and the simulation can be re-ran from these states later.

The topography alteration feature is achieved by controlling an avatar client-side, the user can move its position across the world and increase or decrease the height at a precise point. These topography alteration messages are sent to a topography server whose role is to store the entire list of modifications done to a cell (in order to re-apply them in case the cell is unloaded and reloaded from disk). Once the alteration is performed on a given cell, the data is re-emitted to account for the update. Implementation of save states for the simulation is done simply by saving at an instant the current list of topography alterations and the entire water quantities scalar fields for each cell impacted by the simulation. We also save the current camera configuration. Restoring a state is then accomplished by first setting the saved point of view, then the visualized cells are naturally loaded according to the camera position. Next we also load every cell for which we saved a water quantity array and finally reinitialize these scalar fields with data saved on disk instead of zero. Note that thanks to the restoration of the alteration message list, height fields are loaded and correctly modified on the fly.

D. Application architecture

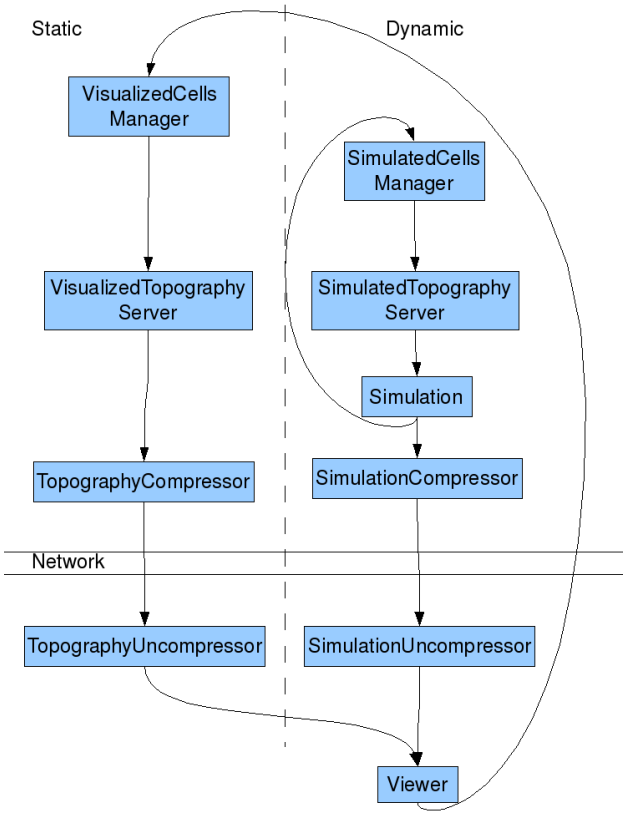

Fig 3: Application architecture

Here is the global architecture of our application. Notice the generic nature of the dynamic simulation integration into an existing large static data visualization framework. One may couple any simulation code with the server side modules and visualize on a distant client the results, as long as the corresponding modules to manage resources and compress/uncompress output data are provided. The simulation core itself doesn't need a major rewrite to be integrated in the model, however thanks to the FlowVR network and software components paradigm, integration is done by developing specific modules to handle data transmission over the network and designing a meaningful and correct synchronization. The LOD algorithm and compression process taking matrices of values as input, it is possible to distantly visualize any information representable as a raster data type. We have done it using height fields, textures and water quantities and it can be extended for example to population density, pollutant concentration, etc...

We now describe the architecture in order to point out the main modules and explain their role. Firstly, the

Viewer is sending the camera position and orientation (known as the frustum describing the 3D region which is visible) to the server so as to receive updated scene geometry afterward. On the server, the VisualizedCellsManager is responsible for computing the list of cell data visible given the current point of view, thus needing actual loading in the cluster memory. Then the VisualizedTopographyServer reads needed cell data from disk and sends them to the TopographyCompressor. As far as the Simulation module is concerned, it receives the list of cells impacted by flood simulation and performs calculations on this data. Then it reports newly impacted cells to the corresponding manager module, and produces output data as a water height values matrix. The CLOD and compression mechanism takes place in the different Compressors modules, right before emission of the scene geometry compressed format on the network. Then specialized Uncompressors modules accomplish data uncompression on the client side part of the application and send it to the Viewer in an almost directly OpenGL displayable format. The Viewer displays the received scene and handles user interaction. Note that each module in relationship with cell data (DataServer, Compressor, Uncompressor...) comes in as many flavors as there is cell data type involved (height fields, textures, etc...), thus allowing further enrichment of the model : one may only develop new modules to compress/uncompress the results of a newly integrated simulation.

E. Synchronization :

At first glance, we chose a combination of classic FlowVR connections to synchronize our application. That is to say FIFO and Greedy connections. FIFO is a good choice when connecting two modules perfectly synchronous, performing different phase of a procedural task when a module needs the output from another before starting its own computation process. For example, we use a FIFO connection between the TopographyServer and corresponding TopographyCompressor module. However, being a real time application, most of the modules must run asynchronously from each other. This is why we use a Greedy connection between the Uncompressors modules and the Viewer, as this module is an interactive one, it should run at maximum frame rate and not suffer from server side latency. SteerGreedy is used wherever needed as previously demonstrated. Particularly, we use this type of connection to control emission of new frustum data to the server, launching heavy computation on the cluster only when the user moves the camera. It is also used later on the connection cycle to allow cascade blocking of server side modules.

In order to minimize network latency, a SteerGreedy connection is also used between the client/server architecture (Compressor/Uncompressor modules). The network objects (synchronizer + filter) are instantiated on the server and when the connection is blocked, no more message will be transferred via the link to the client. Another important aspect is the necessity of the final Greedy connection attached to the Viewer : if we used a SteerGreedy connection at this point, this could result in infinite modules inter-blocking. As soon as the Viewer decides that the content of its output is no more interesting regarding server processing, it would block every single connections thus blocking itself.

F. Cell loading and unloading mechanism

1) Within the simulation

Considering large data visualization using an in-core context, it is acceptable to load the entire data set on PC clusters huge memory. Nonetheless, as far as testing and scalability is concerned, it would be interesting to be able to target lower performances platforms, such as laptops, thus meaning less RAM and incapacity to load all of the data at the same time. Moreover, although we could load everything on the supposedly very powerful server, what is the point if only a small part of the data is needed at some time ? As a result, it is mandatory to adopt a reasonable policy concerning cells loading, and as a matter of fact, unloading them accordingly.

Our approach is based on a simple assertion : at any given time, it is only necessary to load the cells visible from the current frustum. We can enlarge this statement by also loading these visible cells neighborhood, in order to improve latency because neighbors cells might be loaded a few iterations ahead from disk, which is an expensive operation we should anticipate. We receive the viewpoint from the Viewer insofar as we want to simplify cell data using a LOD algorithm, another use of this viewing information is to compute the identifier of the cells that need to be loaded. This is the role endorsed by the VisualizedCellsManager module, taking the point of view as input, it computes the position of the visible cells (those within a given radius around the intersection of the view with the z plane representing the floor). Actual loading/unloading process is done by the different CellDataServers (height fields and textures), at each new iteration, these modules receive the list of cells that should be loaded. By keeping track of the list of cells currently loaded, the module is able to compare these two different lists and deduce the cells that are no longer needed and can be safely unloaded. The module then proceeds by loading the cells it is instructed to provide, via the manager that are not already present in the loaded set.

2) Within the flood simulation

We have decided to store in memory only the cells currently impacted by the course of the simulation, which is a rather obvious behavior. As the flood propagates among the terrain within the simulation process, the

Simulation module provides feedback to the SimulatedCellsManager module, indicating which cells are gradually impacted by the flood. The manager accumulates progressively these cells where the simulation goes on and forwards them to the SimulatedTopographyServer, the module actually loading the terrain data upon which flood is taking place.

IV. Conclusion and perspectives

When we introduced our generic framework at the beginning of this paper, we stressed that as a general template architecture, it could be used to integrate any heavy simulation in a VR environment to perform distant visualization of computationally steered codes. As a result, this work could be used in other domains than what we focused on in the description of our implementation of the model, that is to say flood simulation and visualization of geo-referenced data. To do so, it is necessary to develop LOD algorithms specific to the data type treated, in order to allow compression and distant visualization. A close study about synchronization policies using work on the SteerGreedy is imperative to avoid useless resource consumption and define the most clever application architecture thanks to FlowVR. Currently, we only consider data compression for visualization purpose, but why not trying to extend this concept and using degraded data as input to another distant simulation ? Such simulation would try to make the most of currently available data and provide the best results considering the quality of what we fed it with.

There is a lot of interesting perspectives thanks to the SteerGreedy synchronization policy. For example, we might consider a population density simulation across major axes of movement, inside a richer application. The user could steer the results of this simulation to control whether he wants to visualize this data or not, as these information may not be always useful. Moreover, if a particular semantic is applied to the resulting data of such simulation, the synchronization policy should allow punctual steering by deciding precisely when the results computed on the server have changed enough to require an update on the client.

The application we presented is part of the DALIA project, supported by the ANR, the French national research agency [8].

V. Bibliography

[1] A. Esnard, “Analyse, conception et réalisation d’un environnement pour le pilotage et la visualisation en ligne de simulations numériques parallèles”, 2005

[2] Gropp, “A high-performance, portable implementation of the MPI Message Passing Interface standard”,1996, p.3

[3] J. Allard and V. Gouranton and L. Lecointre and S. Limet and E. Melin and B. Raffin and S. Robert, “FlowVR: a middleware for large scale virtual reality applications”, 2004

[4] S. Madougou, “Intégration des modélisations 3D des sciences de la terre au sein d'environnements de réalité virtuelle à base de grappe de PC”, 2005

[5] Jos Stam, "Stable fluids.", 1999

[6] Mark J. Harris, "Fast fluid dynamics simulation on the GPU"

[7] Viet D. Tran, Ladislav Hluchy, Dave Froehlich, and William Castaings, "Parallelizing flood model for linux clusters with MPI"

[8] http://dalia.gforge.inria.fr/index.php